Abílio Paulo Pinho Neves

INTERACÇÃO REMOTA COM CIRCUITOS IMPLEMENTADOS EM FPGA

Abílio Paulo Pinho Neves

INTERACÇÃO REMOTA COM CIRCUITOS IMPLEMENTADOS EM FPGA

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor Valeri Skliarov, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, co-orientação científica do Professor Doutor Nuno Miguel Gonçalves Borges de Carvalho, Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor António Manuel de Brito Ferrari Almeida
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

Doutor Hélio Mendes de Sousa Mendonça
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Doutor Valeri Skliarov
Professor Catedrático da Universidade de Aveiro (Orientador)

Doutor Nuno Miguel Gonçalves Borges de Carvalho
Professor Associado da Universidade de Aveiro (Co-Orientador)

agradecimentos / acknowledgements

Em primeiro lugar, gostaria de agradecer ao Prof. Valeri Skliarov e ao Prof. Nuno Borges de Carvalho por terem sido excelentes orientadores, sempre disponíveis para ajudar e pelo empenho demonstrado na realização deste trabalho. Gostaria de agradecer também ao João Lima, por ter sido um excelente colega de trabalho, bem como a toda malta da sala 234 pelo bom ambiente de trabalho nestes últimos meses.

Agradeço também á minha família e em especial ao meus pais por todo apoio dado durante o curso. Aos amigos que sempre me acompanharam neste percurso académico, em especial ao Pardal, Lino, Pedro, Soldado, Carlos, João, Camarada,… por toda a amizade nestes últimos anos.

Por fim, deixo esta linha para todos aqueles a quem a minha memória falhou…

Palavras-chave

Estacionamento automático, sistema embutido, *Field Programmable Gate Array*, máquina de estados finitos, *buffer* de prioridade, controlo remoto, simulação visual.

Resumo

Com crescente utilização nos últimos anos de dispositivos como as FPGAs, a construção de módulos reutilizáveis tornou-se importante para a implementação de sistemas cada vez mais complexos. Este tipo de sistemas frequentemente necessita comunicação remota para diversos fins, como por exemplo para controlo remoto, alteração de parâmetros, verificação de estados, entre outros.

Nesta tese foi assim desenvolvido um bloco reutilizável que forneça aos sistemas baseados em FPGA a capacidade de comunicarem sem fios.

Dentro dos sistemas implementados em FPGA, foi analisada a aplicabilidade na prática de modelos avançados de máquinas de estados finitos, para a implementação em *hardware* de algoritmos de controlo modulares, hierárquicos, recursivos e paralelos. Para isso, foi implementado um componente reutilizável, denominado *buffer* de prioridade, que é descrito em detalhe e é sintetizado a partir de uma especificação modular, hierárquica, recursiva e paralela.

Nesta dissertação também é descrito um sistema para controlo automático de um parque de estacionamento. Este sistema composto pelo controlo central e pelo controlo de cada carro são inicialmente ligados directamente, dentro da mesma FPGA, para efeitos de simulação. Para a gestão dos lugares é aplicado o *buffer* de prioridade construído anteriormente.

Por fim, é demonstrado um sistema com controlo remoto, através da implementação da interface sem fios desenvolvida entre o controlo central e o controlo dos carros.

O protótipo do sistema completo foi projectado, implementado em FPGA, avaliado e testado com êxito. Os resultados pertinentes podem ser avaliados através de uma simulação visual apresentada num monitor VGA, evitando assim a necessidade de um ambiente físico dispendioso. Alguns resultados desta tese serão publicados num artigo [1] aceite para apresentação numa conferência internacional.

Keywords Automatic parking, embedded system, Field Programmable Gate Array, finite state machine, priority buffer, remote control, visual simulation.

Abstract

With increasing use in recent years of devices such as FPGAs, the construction of reusable modules has become important for the implementation of increasingly complex systems. Such systems often require remote communication for various purposes, such as remote control, change of parameters, verification of status, etc.

This thesis was developed as a reusable unit that provides the FPGA-based systems the ability to communicate wirelessly.

Within the systems implemented in FPGA, we analyzed the applicability in the practice of advanced models of finite state machines to implement modular, hierarchical, recursive and parallel algorithms in hardware. For this reason it was implemented a reusable component, denominated priority buffer, that is described in detail and is synthesized from a specification modular, hierarchical, recursive and parallel.

In this dissertation is also described a system for automatic control of a car park. This system comprises the central control and the controls of each car are initially directly connected inside the same FPGA, for the simulation effects. For the management of parking slots is applied the buffer priority built previously.

Finally, it is demonstrated a system with remote control, through the implementation of the wireless interface developed between the central control and control of cars.

The prototype of the complete system was designed, implemented in FPGA, evaluated and tested successfully. The relevant results can be evaluated through a visual simulation presented in a VGA monitor, thereby avoiding the need for expensive physical environment. Some results of this thesis will be published in a paper [1] accepted for presentation in international conference.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Enquadramento

A evolução tecnológica verificada nas últimas décadas na área dos circuitos integrados permitiu a implementação de sistemas cada vez mais complexos com milhões de transístores num único circuito integrado (SoC - *System-on-Chip*). Uma consequência lógica desta evolução foi o aparecimento no mercado de dispositivos lógicos programáveis com um enorme potencial como é o caso das FPGAs. Assim, a densidade crescente deste hardware reconfigurável em conjunto com as linguagens de descrição de hardware (HDL -*Hardware Description Language*) permitiu uma grande flexibilidade no desenvolvimento de arquitecturas digitais revolucionando deste modo os sistemas digitais. Desde a primeira FPGA disponível comercialmente pela empresa Xilinx, em 1985, estas tem sofrido grandes desenvolvimentos desde então, quer na sua capacidade lógica, como na sua estrutura, incorporando agora uma grande variedade de componentes digitais embutidos como processadores, blocos de memória, multiplicadores, *transceivers* de alta velocidade e muitos mais. Estas possibilidades juntamente com o seu preço e reconfigurabilidade, tornaram a utilização de um dispositivo como uma FPGA economicamente viável, em cada vez mais aplicações. Deste modo, estas tornaram-se assim uma boa alternativa para a grande maioria dos sistemas embutidos do quotidiano, que actualmente na maioria dos casos estão implementados usando uma solução baseada num microcontrolador, que combina um microprocessador com um conjunto periféricos específicos e deste modo inflexíveis.

1.2 Motivação

Neste contexto, com a crescente utilização de sistemas embutidos baseados em FPGA, estes vão requerer cada vez mais tipos de interfaces para comunicação com o mundo exterior. Apesar de actualmente já se encontrarem disponíveis vários blocos construídos para comunicação de uma FPGA com outros sistemas computacionais e com vários periféricos, todas estas comunicações baseiam-se em ligações com fios. No entanto estes sistemas frequentemente requerem comunicação remota, ou seja, uma ligação sem fios para diversos fins. Uma ligação sem fios pode ser bastante útil nestes sistemas uma vez que pode ser utilizada para várias funções como por exemplo, controlo remoto, melhoramento do comportamento implementado, alteração de alguns parâmetros, verificação de estados do sistema, etc. Deste modo esta tese centra-se na construção de um bloco reutilizável que forneça aos sistemas embutidos baseados em FPGA a capacidade de estes comunicarem remotamente (sem fios), sem a necessidade de recorrer a microcontroladores para esse efeito.

Apesar da possibilidade do uso de outros métodos de design e arquitecturas alternativas, neste trabalho foi feita a utilização de vários tipos estruturas, como modelos avançados de máquinas de estados finitos (FSM), que permitem a implementação em *hardware* de algoritmos de controlo modulares, hierárquicos, recursivos e paralelos, fornecendo deste modo uma grande flexibilidade ao controlo digital por *hardware*. Desta forma é possível aplicar os algoritmos de *software* mais complexos com todas estas propriedades numa implementação puramente lógica, sem a necessidade do recurso a microprocessadores. No entanto, a aplicabilidade de cada propriedade particular referida atrás, como a modularidade, hierarquia, as chamadas recursivas, e paralelismo em *hardware*, precisa de ser testada e avaliada na prática através de um ambiente de trabalho próprio. Para isso, um componente reutilizável do sistema, denominado *buffer* de prioridade, é descrito em detalhe e é sintetizado a partir de uma especificação modular, hierárquica, recursiva e paralela. Com a capacidade das FPGAs actuais é possível utilizar a mesma FPGA para a implementação de um determinado sistema embutido, com a integração de componentes funcionais reutilizáveis, e ao mesmo tempo fazer uma simulação visual num monitor, que através da forma de imagens iterativas e dinâmicas que representam objectos físicos específicos, é possível observar os resultados. Uma abordagem deste tipo permite uma simplificação na prototipagem, avaliação e comparação de diferentes soluções para um determinado problema. Deste modo, este ambiente proporcionado pelas capacidades de uma FPGA, permite resolver problemas do mundo real de uma maneira fácil de entender e usar. Esta técnica é demonstrada através de um exemplo de um sistema automático para controlo de um parque de

estacionamento. Este exemplo foi considerado pois para além deste tipo de sistemas ajudarem a minimizar o tempo e espaço de estacionamento, que nos dias de hoje se tornou um grande problema devido a um crescente número de carros, este pode ser um bom ambiente de teste para os blocos reutilizáveis construídos como a interacção remota e o *buffer* de prioridade. Convém referir que no caso do parque de estacionamento, para a gestão dos lugares ocupados e livre, bastaria por exemplo um simples vector de bits correspondentes a cada lugar do parque. Assim para a ocupação de um lugar bastava procurar sequencialmente no vector o primeiro lugar livre. No entanto, o objectivo é criar um *buffer* de prioridade que pode ser usado, não só neste sistema em particular, mas também em sistema muito mais complexos e sofisticados. Um dos potenciais exemplos de um sistema mais complexo, são as redes de comunicação, onde o número das entradas normalmente não é conhecido á partida e assim essas entradas não podem ser associadas com blocos predefinidos. Para a estrutura deste *buffer* pode-se considerar diferentes tipos de estruturas como listas ligadas de prioridade ou árvores binárias. Contudo, para a demonstração de chamadas recursivas, a estrutura das árvores binárias são mais apropriadas para esse efeito.

1.3 Objectivos

Os objectivos deste trabalho podem assim ser decompostos nas seguintes partes:

- Construção de um bloco reutilizável que permita avaliar a aplicabilidade e eficácia na prática dos modelos avançados de FSMs na implementação de algoritmos modulares, hierárquicos, recursivos e paralelos em *hardware*;
- Implementação e verificação da ocupação e libertação de memória dinamicamente em *hardware*;
- Verificar a capacidade das FPGAs funcionarem como um ambiente que permite a simulação visual dos sistemas implementados e deste modo testar diferentes arquitecturas blocos funcionais reutilizáveis;
- Implementação de um sistema embutido com as características e funcionalidades anteriores;
- Construção de um módulo reutilizável que permita aos sistemas embutidos baseados em FPGA terem capacidade de comunicar remotamente, ou seja, sem fios;
- Verificação e teste da ligação sem fios implementada, que permita definir vários parâmetros e limitações desta;

- Implementação e demonstração de um sistema embutido com controlo remoto.

1.4 Estrutura da Dissertação

Para além deste capítulo introdutório, esta tese está organizada nos seguintes capítulos:

Capítulo 2 - Sistemas embutidos baseados em FPGA - Neste capítulo é feita uma introdução teórica sobre conceitos desde os sistemas computacionais até sistemas embutidos baseados em FPGA, bem como a sua comparação com outras soluções mais tradicionais para os sistemas embutidos. Aqui são evidenciadas as vantagens desta abordagem e como a implementação de processadores embutidos em FPGA podem proporcionar protótipos de *hardware* e *software* com alto desempenho. Por fim, são apresentados os modelos principais das máquinas de estados finitos avançadas, uma vez que são as estruturas de controlo principais deste trabalho.

Capítulo 3 - Ferramentas de desenvolvimento - Neste capítulo é feita uma introdução desde os primeiros dispositivos lógicos programáveis até as FPGAs. São apresentadas as características fundamentais deste dispositivo, desde a sua estrutura, vantagens, aplicações até a sua evolução, e como em conjunto com as LDH e os blocos IP disponíveis revolucionaram os sistemas digitais. Depois disto, também são apresentadas as principais etapas no desenvolvimento de sistemas baseados em FPGA e as ferramentas de suporte que foram utilizadas neste trabalho (ISE e Modelsim). Neste capítulo são ainda apresentadas as placas de desenvolvimento utilizadas neste trabalho (Celoxica RC10 e Nexys 2). Finalmente são descritas as características da ligação sem fios pretendida para interacção remota e deste modo é apresentado o *Transceiver* RF escolhido para esse efeito, o CC1101 da Texas Instruments.

Capítulo 4 - Sistemas implementados - Neste capítulo é feita a descrição detalhada da implementação prática dos sistemas embutidos construídos, mais especificamente, o *buffer* de prioridade construído através de um estrutura baseada numa árvore binária, um sistema próprio para teste deste tipo de *buffer* e finalmente um sistema embutido para controlo automático de uma parque de estacionamento.

Capítulo 5 - Interacção remota - Neste capítulo é feita a descrição detalhada da implementação prática do módulo que permite a interface com o *transceiver* RF

escolhido para a respectiva comunicação sem fios. Para isto, é descrita uma solução que utiliza uma decomposição hierárquica do tipo "*bottom-up*" desde a camada mais a baixo que gera o relógio (SCLK) para o protocolo SPI que faz a comunicação com o *transceiver* RF, até a camada mais a cima "*top-level*" que recebe e envia os dados que o sistema precisa de transferir para outro remotamente (sem fios). No final é implementada testada a interacção remota no sistema do parque de estacionamento do capítulo anterior.

Capítulo 6 - Conclusões e trabalho futuro - Neste capítulo são apresentadas as conclusões principais desta dissertação fazendo uma análise global aos resultados e objectivos atingidos. No final são apresentadas algumas possibilidades para um eventual trabalho futuro.

Apêndice A - Neste apêndice são apresentados algumas características do *transceiver* RF CC1101 que permitem compreender melhor a implementação descrita no capítulo 5.

Capítulo 2

Sistemas embutidos baseados em FPGA

Sumário

Neste capítulo é feita uma introdução teórica sobre conceitos desde os sistemas computacionais até sistemas embutidos baseados em FPGA, bem como a sua comparação com outras soluções mais tradicionais para os sistemas embutidos. Aqui são evidenciadas as vantagens desta abordagem e como a implementação de processadores embutidos em FPGA podem proporcionar protótipos de hardware e software com alto desempenho. Por fim, são apresentados os modelos principais das máquinas de estados finitos avançadas, uma vez que são as estruturas de controlo principais deste trabalho.

2.1 Sistemas computacionais

Os sistemas computacionais podem ser divididos essencialmente em dois grupos, consoante a sua flexibilidade e a sua aplicação. Num grupo encontram-se os sistemas computacionais de uso geral, onde o exemplo mais comum é o computador pessoal (PC). Estes são bastante flexíveis, podendo executar uma grande variedade de aplicações (programas), com a ajuda de um sistema operacional que gere os recursos de sistema. Em contraste, no outro grupo encontram-se os sistemas computacionais especializados a uma determinada aplicação, particularmente os sistemas embutidos. Normalmente estes sistemas só têm uma aplicação que a executam permanentemente. No entanto, um sistema computacional de uso geral, ou um sistema embutido, apresentam princípios básicos de operação essencialmente iguais e assim uma arquitectura física semelhante (*hardware*), pois a sua diferença principal consiste na aplicação a que se destinam,

reflectindo-se assim no seu software. Deste modo, um sistema computacional do ponto de vista do *hardware* pode ser considerado como uma combinação de um processador que executa os programas (de *software*), memória que pode ser de vários tipos para armazenar os dados e os programas, e por dispositivos de entrada e saída (I/O) que permitem a comunicação com o mundo exterior como por exemplo um teclado e um monitor, respectivamente.

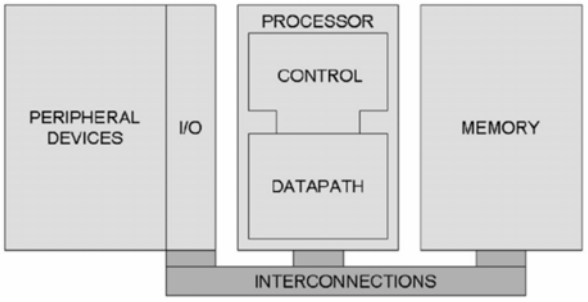


Figura 2.1: Sistema computacional básico[2].

O processador é a parte mais importante de um computador, pois é este dispositivo que processa e manipula os dados, ou seja a informação, segundo uma certa ordem que é o que constitui um programa. Um microprocessador também conhecido por vezes como um CPU (unidade central de processamento) é um processador implementado em um único circuito integrado, exemplos deste tipo são a série Pentium da Intel, Freescale / IBM, MIPS, ARM e o Sun SPARC, entre outros. Um microcontrolador é um processador, memória e alguns dispositivos de I/O implementados dentro de um mesmo circuito integrado, e destinado essencialmente para uso em sistemas embutidos. Deste modo a comunicação entre estes é feita do mesmo circuito integrado. A gama de microcontroladores disponíveis é muito larga, desde as pequenas PICs e AVRs a processadores PowerPC com I/O embutidos.

2.1.1 Sistemas embutidos

Este grupo destina-se principalmente para sistemas com exigências de baixo custo e consumo de energia, com poucos componentes, uma resposta em tempo real e uma co-existência entre *hardware* e *software*. Apesar destes sistemas computacionais serem menos conhecidos ao utilizador comum, representam um mercado muito maior, pois este tipo de sistemas, podem ser encontrados escondidos em todo tipo de aplicações do dia-a-dia: -em casa (TVs, DVDs, máquinas de lavar, consolas de jogos); -nos escritório (impressoras, WLAN); -nos carros (controlo do motor, travões ABS, sistemas de segurança); -nos aviões (navegação, auto piloto); -na indústria (controlo embutido para

executar vários processos tecnológicos); -na robótica; -no bolso (telemóveis, dispositivos mp3, PDAs, cartões electrónicos). Em geral, os sistemas embutidos costumam ser mais simples e com menor capacidade de processamento que um computador pessoal (PC), como por exemplo nas suas aplicações mais comuns, nas TVs, maquinas de lavar, etc. No entanto estes podem ser muito mais complexos e com muito maior capacidade de processamento que um simples PC, como por exemplo um sistema distribuído de controlo de um avião. Um sistema embutido normalmente não contém um sistema operacional e raramente e possível instalar um novo *software*, contendo o seu software numa memória não volátil, ou seja permanente. Em muitas situações um sistema embutido pode ser usado para substituir um circuito electrónico específico, pois o uso de um microprocessador embutido permite que a resolução do problema seja feita via *software* e não por *hardware*, o que para uma aplicação complexa facilita a sua construção e manutenção.

Nas últimas quatro décadas assistiu-se a uma grande evolução da tecnologia na área dos circuitos integrados, seguindo a lei de Moore [24], que indica que o número de transístores num único circuito integrado tem um crescimento exponencial (duplica a cada 18 a 24 meses). Na figura 2.2, é possível observar um exemplo deste crescimento exponencial verificado na evolução do número de transístores dos microprocessadores da Intel ao longo das últimas quatro décadas.

Microprocessor	Year of Introduction	Transistors
4004	1971	2,300
8008	1972	2,500
8080	1974	4,500
8086	1978	29,000
Intel286	1982	134,000
Intel386™ processor	1985	275,000
Intel486™ processor	1989	1,200,000
Intel® Pentium® processor	1993	3,100,000
Intel® Pentium® II processor	1997	7,500,000
Intel® Pentium® III processor	1999	9,500,000
Intel® Pentium® 4 processor	2000	42,000,000
Intel® Itanium® processor	2001	25,000,000
Intel® Itanium® 2 processor	2003	220,000,000
Intel® Itanium® 2 processor (9MB cache)	2004	592,000,000

Figura 2.2: Evolução do número de transístores[3].

Assim, com tamanho decrescente do transístor e com uma densidade de lógica crescente, os sistemas digitais evoluíram muito, aumentando as suas competências e velocidades ficando assim cada vez mais complexos. Deste modo, com este avanço rápido e

contínuo desta tecnologia dos circuitos integrados, a implementação de sistemas embutidos altamente integrados num só circuito passou a ser uma realidade, dando origem a um novo conceito de sistema integrado SoC (*System-on-Chip*). Com isto, é possível encontrar-se implementados todos os componentes de um sistema computacional complexo dentro do mesmo circuito integrado, como por exemplo, processadores, memoria e controladores de comunicação I/O. Este tipo de abordagem apresenta muitas vantagens devido a interacção muito próxima entre os diferentes componentes do sistema pois permite um aumento a velocidade do sistema, e uma diminuição do tamanho, custo e consumo de energia do sistema, correspondendo deste modo ás exigências de muitos tipos de aplicações. Assim este tipo de circuitos integrados estão cada vez mais presentes á nossa volta, em vários dispositivos e serviços omnipresentes que utilizam estes sistemas. Deste modo, os sistemas embutidos são o segmento com maior crescimento na indústria dos sistemas computacionais, uma vez que assim, podem ser aplicados em mais áreas.

2.2 Sistemas de controlo embutido

Esta evolução dos circuitos integrados nas últimas quatro décadas revolucionou os sistemas electrónicos que eram baseados em componentes analógicos como transístores, amplificadores operacionais, resistências, condensadores e indutâncias. Apesar destes sistemas oferecerem processos paralelos, apresentavam problemas de desvio de parâmetros provocados pela temperatura e pelo envelhecimento. O microprocessador da Intel 4004 foi a primeira plataforma digital que era possível configurar através de *software*. Assim com a chegada dos componentes digitais cada vez mais diversificados, grande parte do controlo de embutido passou para o mundo digital. Dentro destes sistemas de controlo embutido, existem actualmente várias plataformas digitais com vantagens e limitações inerentes a cada uma. As principais plataformas digitais contemporâneas que podem ser escolhidas para implementação de um controlador digital são[4]:

- Microprocessadores;
- Microcontroladores;
- ASSPs (*Application Specific Standard Product*);
- FPGAs;

Assim, a escolha de uma destas plataformas em detrimento das outras soluções será feita em função das características mais apropriadas para as necessidades do controlador digital que se pretende implementar. Deste modo, consoante a aplicação final,

as necessidades principais de um controlador digital num sistema embutido que contribuem para a escolha de uma das plataformas, podem ser distintas como as seguintes [4]:

- Consumo de energia;
- Preço;
- Tempo de resposta em tempo real;
- Quantidade da produção;
- Ferramentas de programação;
- Livrarias e código portável e reutilizável;

2.2.1 Microprocessador

O microprocessador foi a primeira plataforma digital com a capacidade de ser configurável através de *software*, mudando assim a metodologia dos sistemas digitais como nenhum outro componente e ainda continua a ser a escolha como controlador digital para várias áreas de aplicação. Existem muitas aplicações em tempo real, com elevadas taxas de actualização que requerem que o microprocessador seja programado no seu *assembly* nativo. Embora a maioria dos microprocessadores comerciais usados hoje sejam para aplicações de dados, há núcleos de microprocessadores embutidos em microcontroladores para aplicações de controlo em tempo real. Em sistemas de controlo digital, a utilização do processador é feita através de interrupções para processamento em tempo-real. Existem interrupções para diferentes objectivos dentro do mesmo controlador. Cada interrupção vai ser gerada em função do tempo de execução de uma determinada tarefa.

Na figura 2.3 é possível observar o modo de funcionamento de um sistema de controlo digital através das rotinas de serviço de interrupção. Assim sempre que é activada uma interrupção, o microprocessador executa o algoritmo respectivo e actualiza os seus resultados. A grande desvantagem deste tipo de abordagem deve-se ao facto de o microprocessador (de uso geral) poder executar apenas uma instrução de cada vez, e assim este pode não ser suficientemente rápido para atender a múltiplas interrupções de diferentes aplicações de um controlador, não podendo assim implementar controladores mais complexos.

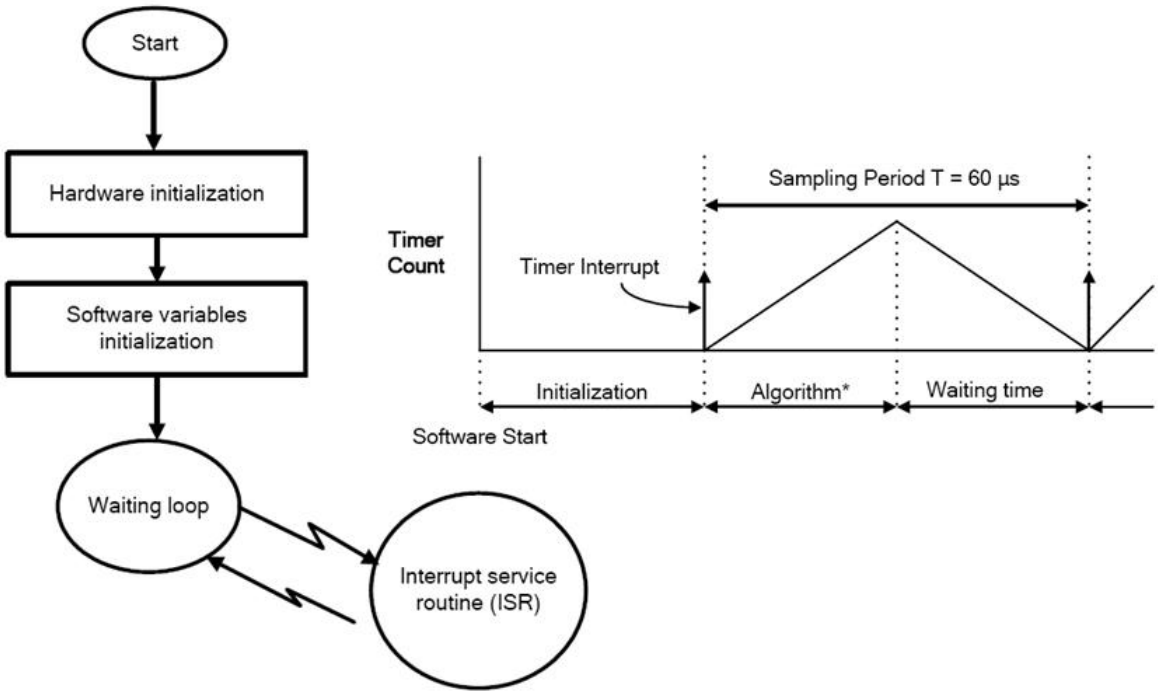


Figura 2.3: Sistema de controlo baseado em rotinas de serviço de interrupção (ISR)[4].

2.2.2 Microcontrolador

Figura 2.4: Microcontrolador PIC.

Os microcontroladores apresentam a combinação do processador com vários periféricos no mesmo circuito integrado, fornecendo assim mais soluções para controladores em sistemas embutidos, pois permitem criar sistemas mais complexos com poucos componentes, uma vez que estes já incorporam periféricos que antes eram externos ao circuito integrado, oferecendo maior integração e assim menor custo e tamanho que os microprocessadores.

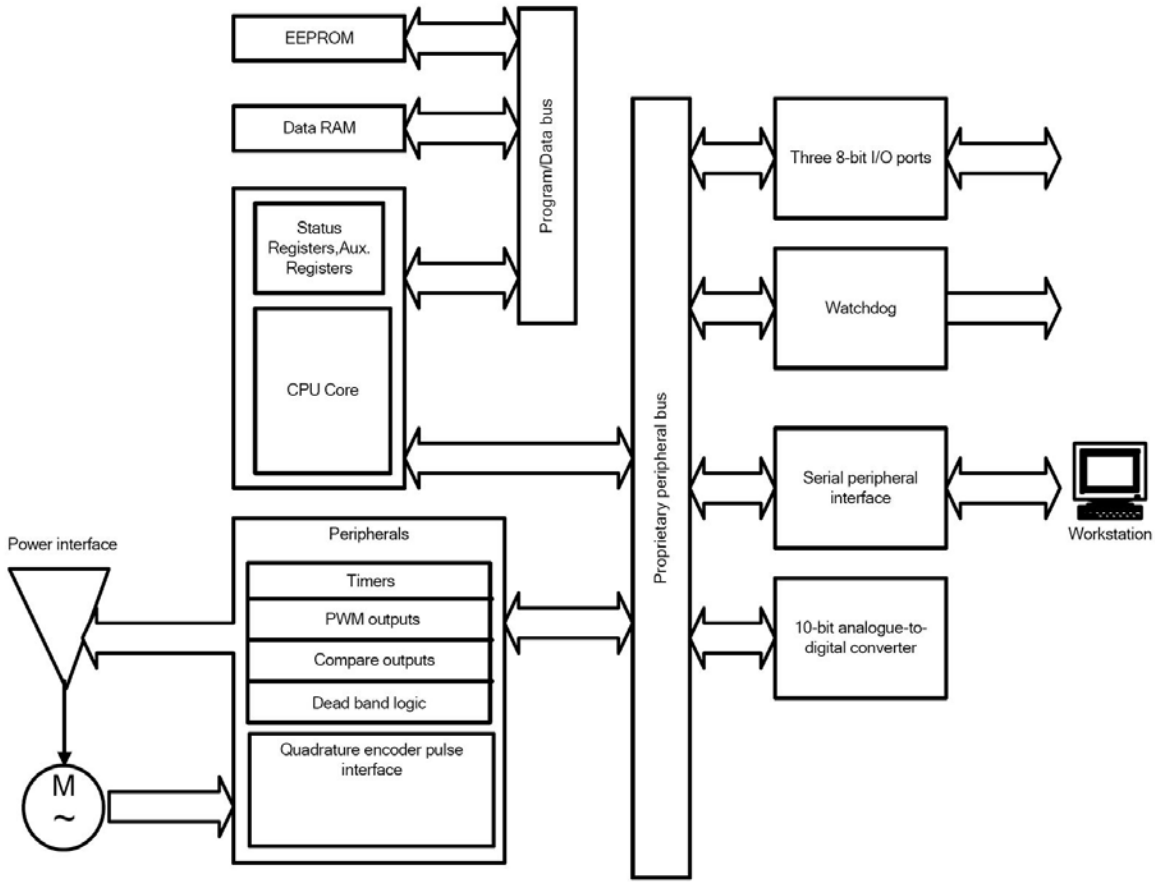


Figura 2.5: Controlador de um motor baseado num microcontrolador[4].

Como no caso dos microprocessadores, as tarefas vão ser divididas consoante as taxas de actualização necessárias. Assim, são utilizadas linguagens de programação de software para as tarefas com baixas taxas de actualização, enquanto que para tarefas com taxas de actualização elevadas e deterministas é usado a linguagem assembly nativa do microcontrolador respectivo. Deste modo, são usados compiladores e linguagens assembly para converter programas de linguagens de alto-nível no respectivo código máquina, de modo a ser armazenado na memória embutida do microcontrolador.

Na figura 2.5, pode ser visto um exemplo muito comum para a utilização de um microcontrolador, como é o caso de o controlo de um motor. Para a maioria das aplicações o uso de um microcontrolador é a solução mais adequada, no entanto este tem como grande desvantagem ter um número fixo de periféricos. Embora exista uma gama extensiva de microcontroladores com números e tipos diferentes de periféricos disponíveis no mercado, nem sempre é possível encontrar um que encaixe perfeitamente em todas as exigências de uma determinada aplicação.

2.2.3 ASSPs (Application Specific Standard Products)

Um ASSP (Application Specific Standard Product) é um circuito integrado que implementa como o próprio nome indica, uma função específica que pode ser aplicada numa extensa gama de sistemas, atraindo assim um grande mercado. Estes dispositivos são usados em todo tipo de indústrias, desde a automóvel até a de telecomunicações. Exemplos de ASSPs são circuitos integrados para controlo de motores e para codificação e descodificação de áudio e vídeo. Na próxima figura pode ser visto um exemplo destes circuitos.

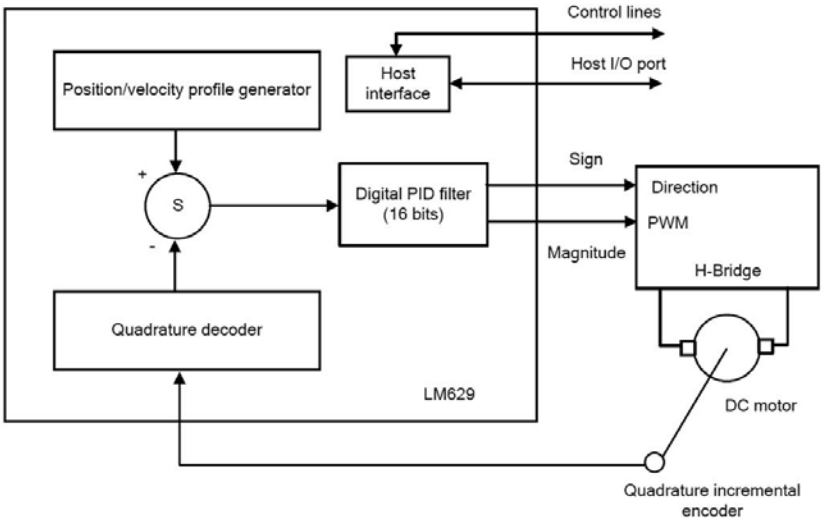


Figura 2.6: ASSP LM629 para controlo de um motor de DC[4].

Normalmente, a funcionalidade de um ASSP é especificada através da sua palavra de controlo. Alguns ASSPs são mais configuráveis e para isso fornecem endereços, dados e controlo de barramento para serem ligados a um processador do sistema que os pode assim controlar.

2.2.4 FPGAs

Por fim, existem as FPGAs que fornecem a possibilidade de no mesmo circuito integrado incorporar um processador e toda a lógica necessária e específica a uma determinada aplicação. Deste modo este dispositivo combina as vantagens das três plataformas digitais descritas anteriormente. Apesar disto, os microcontroladores ainda são muitas vezes preferidos em relação as FPGAs devido ao preço e consumo de energia, no entanto estas tem vindo a ganhar o seu espaço pois oferecem muitas vantagens, como por exemplo a portabilidade do seu código através de vários fornecedores, bem como a possibilidade de reutilizar código, livrarias e ferramentas de programação de baixo custo.

Para além disso, a existência de uma grande quantidade de núcleos de propriedade intelectual permite aumentar de uma forma bastante significativa a produtividade, pois permite uma prototipagem rápida dos sistemas.

Figura 2.7: Aspecto de uma FPGA.

Estes dispositivos lógicos programáveis que tradicionalmente tinham pouca capacidade, agora são capazes de suportar sistemas lógicos de grande capacidade, permitindo assim, implementar sistemas embutidos com controladores de elevada complexidade utilizando apenas uma FPGA. Assim, com a elevada capacidade que cada vez mais oferecem e com um preço cada vez mais acessível, estas oferecem uma grande flexibilidade no desenho dos sistemas embutidos digitais devido á sua reconfigurabilidade inerente. Deste modo, sistemas baseados em FPGA são cada vez mais economicamente viáveis para mais aplicações, pois estes sistemas integram actualmente além de tradicional lógica digital, processadores, memórias, multiplicadores e interfaces de comunicação em um único circuito integrado, como veremos mais a frente com mais detalhe, visto que este trabalho baseia-se nesta plataforma digital.

2.3 Processadores embutidos em FPGA

Actualmente, a maioria dos fornecedores de FPGAs já disponibilizam processadores embutidos nestas (*hard* IP) fisicamente em silicone ou processadores que podem ser implementados através da sua lógica programável (*soft* IP). Com isto, é possível ter um processador com blocos de lógica digitais convencionais dentro do mesmo circuito integrado, de maneira a combinar *software* e *hardware*, fornecendo toda a flexibilidade no controlo de uma determinada aplicação.

Muitos algoritmos que são difíceis de implementar em HDL e que não são críticos em relação ao tempo de resposta podem ser implementados através do processador embutido na FPGA (*software*), enquanto que para as restantes tarefas de controlo

Processor name	Type/Bits	Interface bus	FPGA vendor
MicroBlaze™	Soft/32	IBM Coreconnect	Xilinx
NIOS®	Soft/32	Avalon	Altera
LatticeMico32	Soft/32	Wishbone	Lattice
CoreMP7	Soft/32	APB	Actel
ARM Cortex-M1	Soft/32	AHB	Vendor independent
LatticeMico8	Soft/8	Input/Output ports	Lattice
Core8051	Soft/8	Nil	Actel
Core8051s	Soft/8	APB	Actel
PicoBlaze™	Soft/8	Input/Output ports	Xilinx
PowerPC	Hard/32	IBM Coreconnect	Xilinx
AVR	Hard/8	Input/Output ports	Atmel

Figura 2.8: Lista de alguns processadores embutidos em FPGAs[4].

mais sensíveis aos tempos de resposta são implementadas em *hardware*, através dos recursos lógicos programáveis da FPGA. Para esta combinação de *hardware* e *software*, também estão disponíveis pelos fornecedores de FPGAs, um conjunto de ferramentas que facilitam este tipo de implementações. Com esta possibilidade, o uso de dispositivos como a FPGA ganhou uma nova dimensão pois dão a liberdade dividir os projectos num fluxo de *software* sequencial e em lógica digital (*hardware*) paralela. Existem diferentes processadores de 8 e 32 bits disponíveis pelos fornecedores de FPGA como se pode verificar na figura 2.8, de modo a estes serem utilizados de forma rápida e intuitiva e assim reduzir o tempo de desenvolvimento dos projectos.

Como já vimos anteriormente, este tipo de processadores embutidos em FPGAs estão disponíveis de duas formas, como blocos embutidos fisicamente dentro da FPGA (*hard* IP) ou implementado nos recursos lógicos das FPGAs (*soft* IP). Estes também podem ser escolhidos consoante a aplicação, existindo processadores de ambos tipos de 8 e 32 bits. No entanto o modo de programação destes tipos de sistemas é totalmente semelhante. Como exemplo dos processadores *hard* são os PowerPC que se encontram na família das FPGAs Virtex da Xilinx e o microcontrolador AVR fornecido pela Atmel.

Processadores do tipo *soft* são implementados através do uso dos recursos lógicos de uma FPGA, podendo assim ser implementados pelo utilizador, consoante a necessidade

de uma determinada aplicação. Com este tipo de abordagem, como a única restrição são os recursos lógicos da FPGA respectiva, é possível implementar vários processadores no mesmo circuito integrado, abrindo ainda mais o leque de aplicações deste tipo de plataforma digital, como por exemplo o estudo e desenvolvimento de arquitecturas multi-processador. Entre os processadores de 8 bits, um dos exemplos mais conhecidos é o PicoBlaze fornecido pela Xilinx cujo diagrama de blocos pode ser visto na figura 2.9. Estes tipos de processadores são disponibilizados na forma de *soft* cores sintetizáveis.

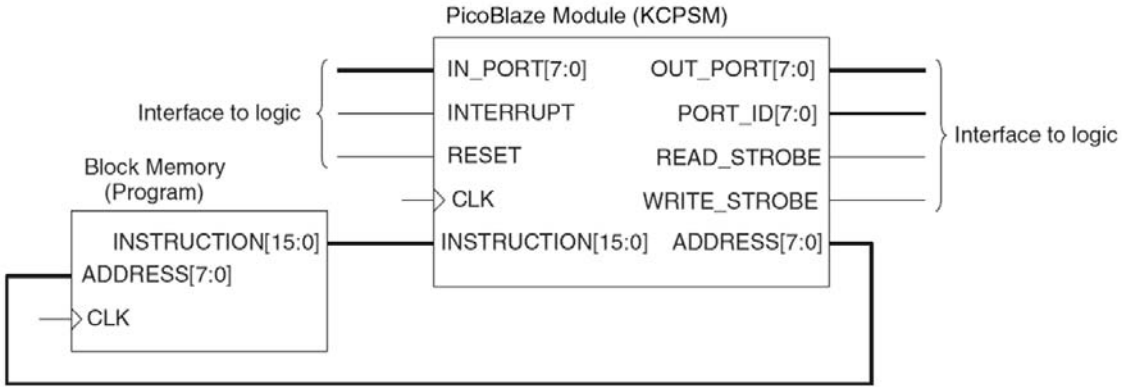


Figura 2.9: Diagrama de blocos do PicoBlaze[5].

Este processador do tipo RISC está disponível na forma de código VHDL sintetizável ou como um IP *core* livre pré sintetizado [5]. Este microcontrolador ocupa uma pequena percentagem dos recursos lógicos de uma FPGA e utiliza os blocos de memória RAM embutida nesta para o armazenamento dos programas e dos dados. Apresenta também um porto de entrada e saída de 8 bits, bem como uma entrada para responder a interrupções externas.

Dentro dos processadores (softs) de 32 bits, é possível destacar também dois dos fornecedores maiores de FPGAs, são os processadores NIOS da Altera e o MicroBlaze da Xilinx. Estes processadores sendo mais complexos usam naturalmente mais recursos da FPGA que o anterior. O resto da FPGA pode ser usado tanto para implementar outra lógica digital, como para implementar outros processadores (*multi-core*). Na figura 2.10 é possível observar o diagrama de blocos do processador MicroBlaze de 32bits RISC, que contém um conjunto de instruções optimizadas para sistemas embutidos [25].

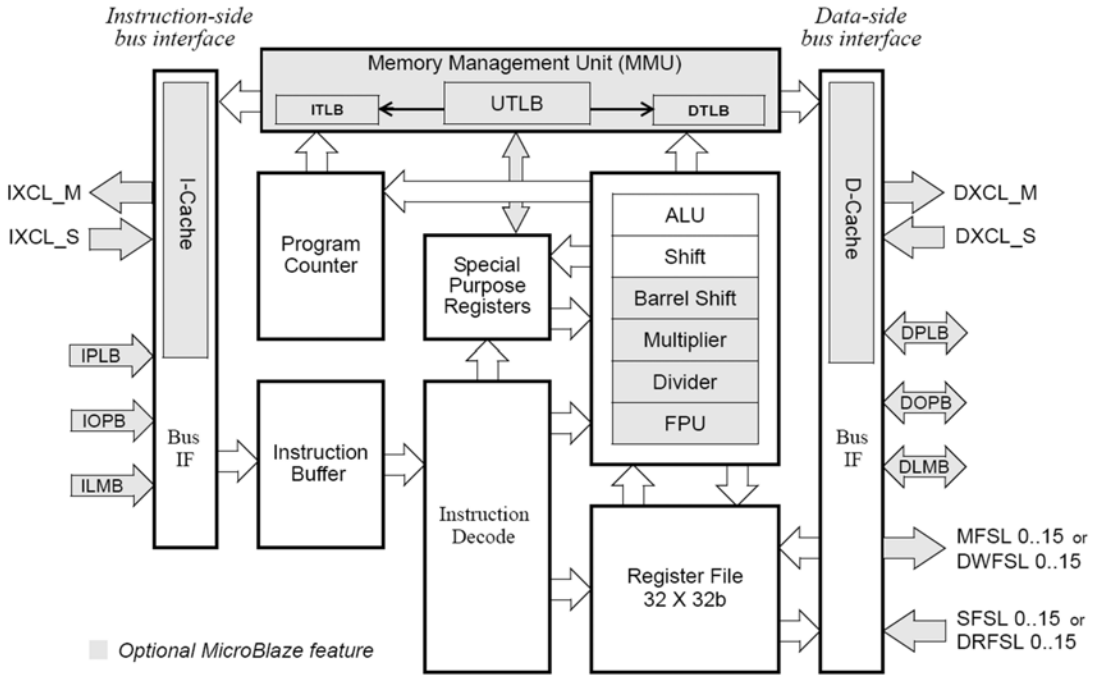


Figura 2.10: Diagrama de blocos do MicroBlaze[6].

Este pode utilizar os blocos de memória embutidos na FPGA e também memória externa para armazenar dados e programas. O MicroBlaze apresenta várias opções de configuração, podendo ser utilizado desde para um controlador de um pequeno sistema embutido até como um computador embutido de alto desempenho capaz de correr um sistema operativo como o Linux. Este contém como opção inicial de implementação uma unidade de vírgula flutuante (FTU) compatível com o standard IEEE-754 permitindo assim adições, subtracções, multiplicação, divisão e comparação com vírgula flutuante, fundamental para muitos tipos de aplicações. Deste modo este processador é bastante personalizável, pois e possível especificar quais os módulos de hardware que se pretende incluir, utilizando assim apenas os recursos necessários.

2.4 Co-design de Hardware e Software

Actualmente, as FPGAs são utilizadas numa ampla variedade de contextos práticos, no entanto com esta possibilidade de implementação de processadores embutidos, protótipos de *hardware/software* (co-design e de co-simulação) ainda são as áreas onde a aplicação de FPGAs é mais eficiente e promissora. Por este motivo, muitos estudos actuais são dedicados a fazer progressos nesta direcção.

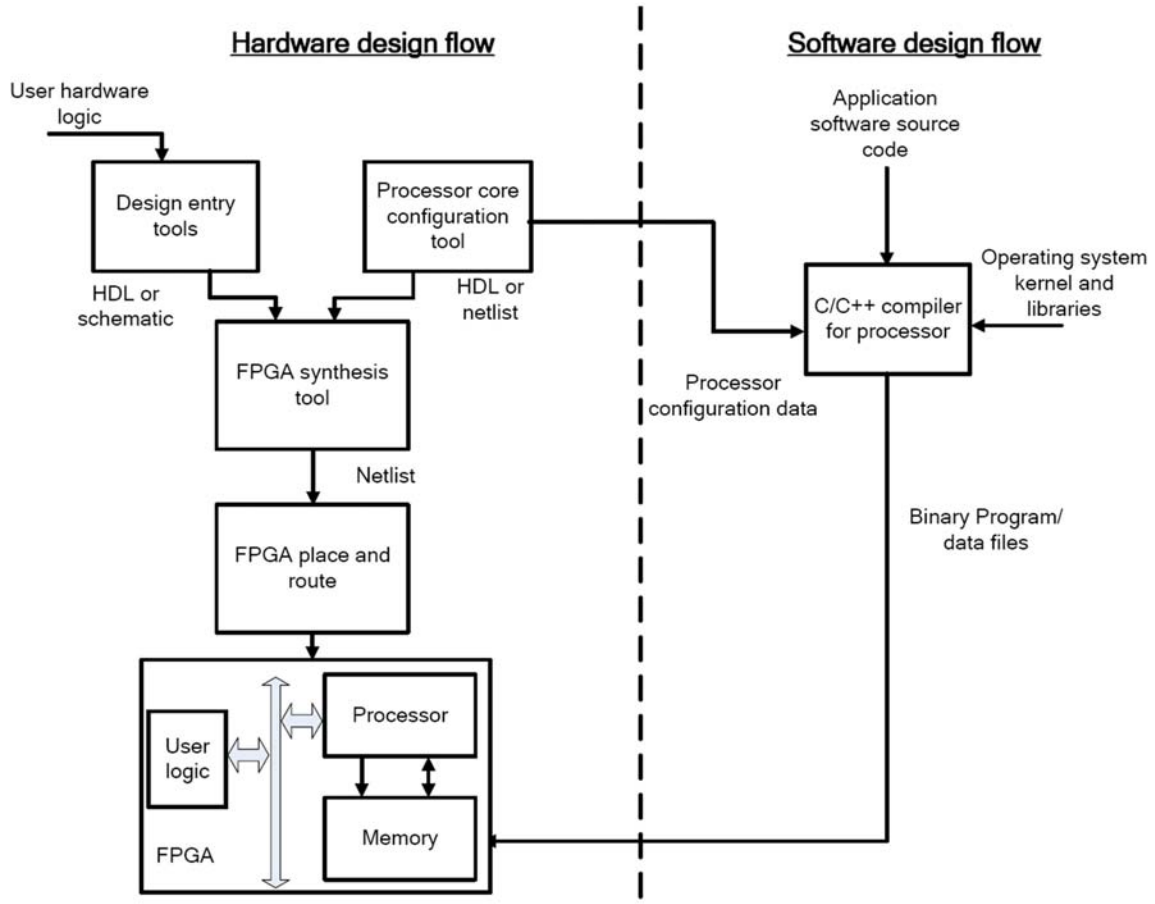


Figura 2.11: Diagrama de fluxo no desenvolvimento de *hardware* e *software*[4].

Para projectar um sistema deste tipo, todas os algoritmos em *hardware* e de *software* devem ser codificados e testados independentemente como se pode verificar na figura 2.11. O fluxo de *software* associa as bibliotecas ao código fonte para criar um ficheiro executável para o processador usar. O código do programa e os dados podem ser armazenados na memória interna da FPGA (*Block RAMs*) ou num dispositivo de memória externo caso o código do programa seja demasiado grande. O fluxo de *hardware* combina ficheiros de formato de intercâmbio electrónico (EDIF) do processador soft, com o código da lógica HDL e prepara depois da síntese uma netlist final do sistema completo. Deste modo, é possível o desenvolvimento completo de um sistema embutido de controlo. Uma das ferramentas possíveis para este tipo construção é a ferramenta EDK disponibilizada pela Xilinx.

A grande vantagem associada á utilização de *software* convencional nas simulações reside na sua flexibilidade, tal como acontece nos computadores de uso geral, pois com estes é possível alterar os parâmetros da simulação facilmente permitindo assim analisar o comportamento de diferentes arquitecturas. No entanto este tipo de abordagem,

apresenta como grande desvantagem um elevado tempo de simulação. Já na simulação (emulação) completa em *hardware* acontece exactamente o contrário, pois permite um significativo aumento de velocidade nas simulações, no entanto a flexibilidade é contudo reduzida. Deste modo, para tarefas que não requerem elevada capacidade de processamento de dados, estas podem ser implementadas totalmente em *software*, ou seja, todo o algoritmo pode ser executado por um microprocessador. No entanto, para tarefas que necessitam melhor desempenho, que ocupam a maior parte do tempo do processador devido a um elevado processamento de dados (paralelo), podem ser implementadas num *hardware* específico (co-processador específico), podendo deste modo aumentar consideravelmente o desempenho global do sistema.

Assim um co-simulador de *hardware-software* (HW-SW) é essencial para avaliações precisas de performance de sistemas reconfiguráveis. Vários simuladores foram desenvolvidos de modo a estender a arquitectura de um microprocessador com *hardware* reconfigurável. Uma proposta apresentada em [7] para a ligação entre o processador e a unidade reconfigurável é baseada no controlo da comunicação pelo *bus*, onde a unidade reconfigurável não tem acesso directo ao processador, não sendo necessário acrescentar nenhuma instrução ao processador. No entanto esta solução apresenta alguns problemas, devido a largura da banda e constrangimentos de latência imposta pelo *bus*.

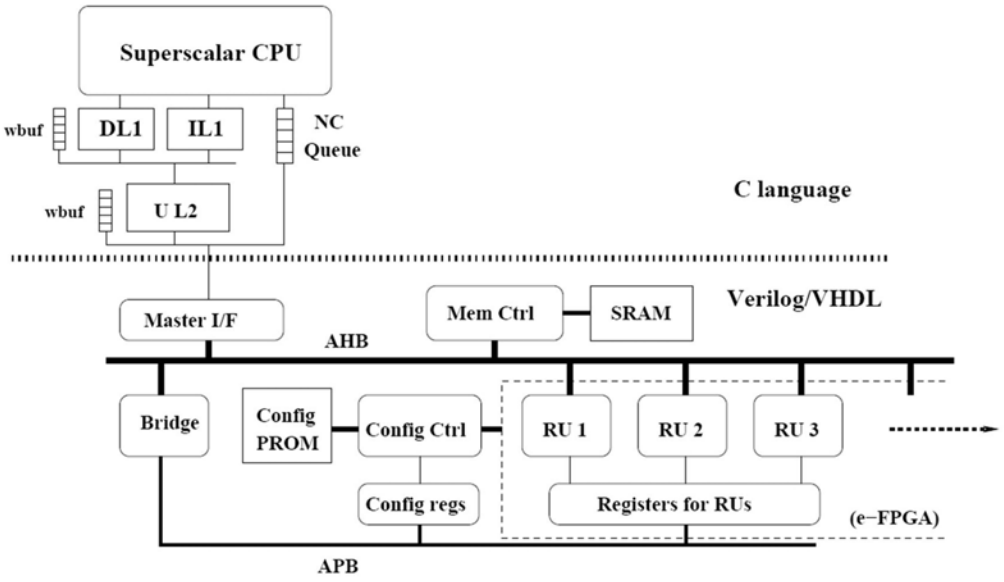


Figura 2.12: Modelo do sistema proposto em[7].

Contudo, com o aparecimento do conceito SoC, onde o bus está incorporado no

mesmo circuito integrado que o processador e outros componentes do sistema, estes constrangimentos foram resolvidos. Deste modo, para SoC esta é das soluções mais flexíveis e modulares, pois o processador funciona assim como *master* e a unidade reconfigurável como *master* ou *slave*, permitindo também que o número de unidades reconfiguráveis activas possa ser variado dinamicamente. O diagrama de blocos da arquitectura utilizada em [7] que se pode observar na figura 2.12, inclui uma unidade de processamento central, modelada em linguagem C, funciona como *master* no bus, enquanto que o controlador de memória e as várias unidades reconfiguráveis funcionam como *slaves* no bus. Estas últimas bem como o bus foram utilizados modelos em linguagem HDL. Tal como era esperado, este sistema inteiro demonstra um significativo aumento de velocidade quando comparado com a pura implementação em *software*, em duas aplicações, nomeadamente, na multiplicação de matriz e na compressão de Lempel-Ziv como se pode observar com mais detalhe em [7].

Em [8] também é proposto um método para de co-simulação em *hardware/software* utilizando um sistema convencional e uma FPGA de maneira a combinar flexibilidade do *software* e desempenho do *hardware*. O simulador é executado assim numa máquina convencional com uma ou várias partes do simulador são implementadas em FPGA.

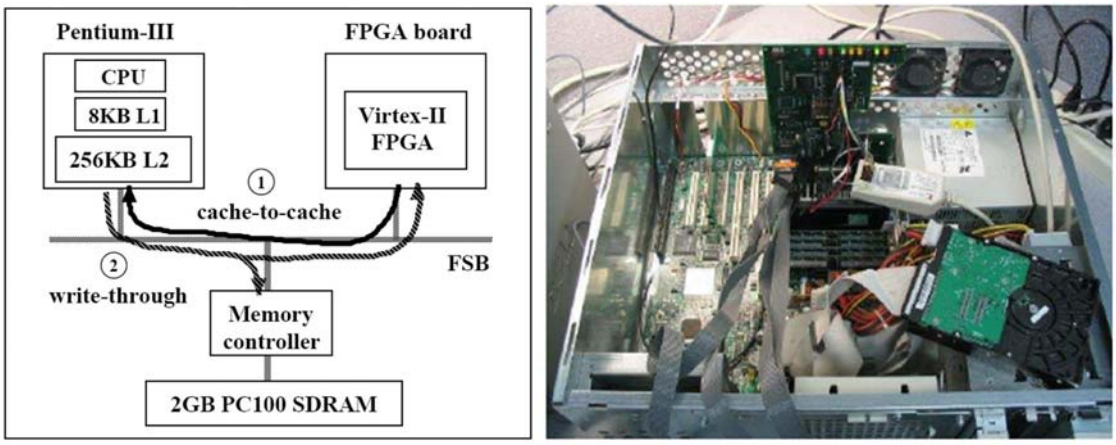


Figura 2.13: Modelo do sistema proposto em [8].

Os resultados desta proposta indicam há ainda alguns problemas associados a este método, que precisam ser resolvidos, nomeadamente o impacto negativo do uso de um bus como meio de comunicação como já tinha acontecido na proposta anteiror. Devido a este facto, não é fácil ter as vantagens da implementação em *hardware*, excepto se todo simulador seja implementado em hardware. Apesar disto, daqui foi possível concluir que a FPGA ainda assim poderá ser útil como era previsto para funções

que necessitam longos tempos de execução em *software*, como por exemplo funções DSP(*Digital Signal Processing*), a Transformada de Fourier (FFT) e *Inverse Discrete Cosine Transfer* (IDCT). Assim, estas funções podem ser implementadas em *hardware* de modo a aumentar o desempenho do sistema, e diminuir o tempo de simulação. Daqui também foi concluído, que a utilização de uma FPGA pode ser uma boa alternativa em pesquisa e investigação da arquitectura *multi-core*, para melhorar o tempo de simulação uma vez que a simulação *multi-core* leva muito mais tempo em relação a simulação de um único *core*.

Em [9] é também apresentada uma metadologia que permite acelarar a simulação de sistemas computacionais usando para esse efeito também FPGAs. Esta técnica consiste em dividir o simulador em duas partes, a primeira simula a função do sistema computacional respectivo. A segunda parte é um modelo facilmente modificável, que prever o desempenho e algumas tarefas, e que implementa assim a maior parte das tarefas paralelas em *hardware*. Deste modo, é eliminada a maior parte da dificuldade e complexidade em simular construtores paralelos numa plataforma sequencial.

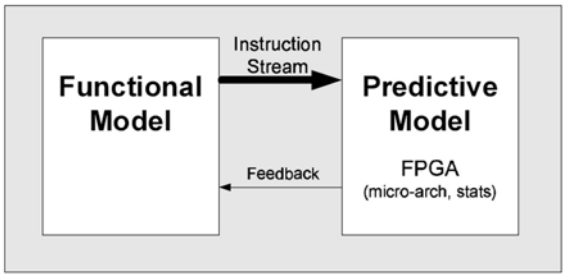


Figura 2.14: Modelo do sistema proposto em [9].

Os resultados obtidos com este método, mostram que com este protótipo, o desempenho do sistema melhora consideravelmente (desde 2 a 7 vezes melhor). Assim, com o uso de apenas uma FPGA é possível simplificar o simulador e atingir um alto desempenho do mesmo.

No artigo [10] é comparada também a simulação em *software* com a co-simulação híbrida (*software* com *hardware* baseado em FPGA) para simulações de sistemas multiprocessador como se pode observar na figura 2.15.

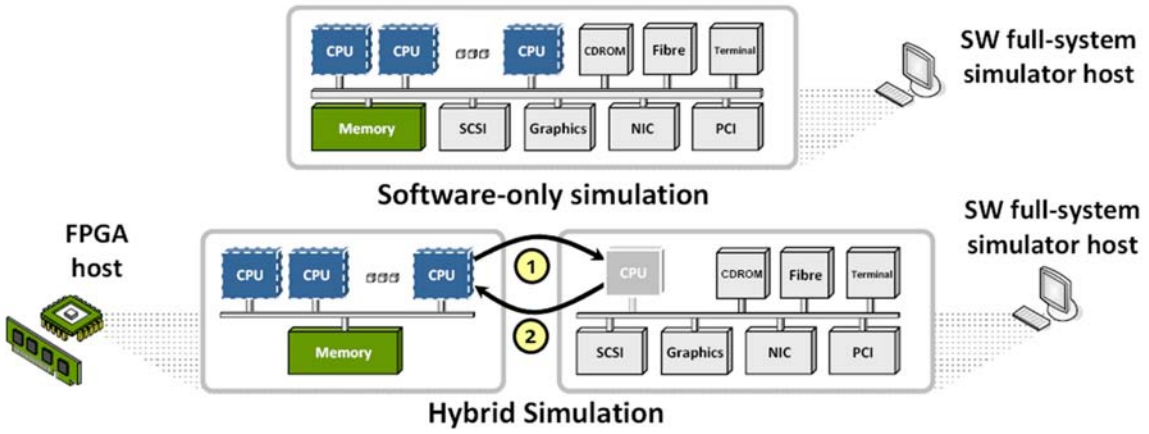


Figura 2.15: Simulação em software e a co-simulação híbrida [10].

Este tipo de simulações tem vindo a ganhar importância, devido crescente adopção de sistemas computacionais altamente concorrentes através deste tipo de arquitecturas. Assim a pesquisa e investigação direcciona-se agora para mecanismos que permitam o aumento da programabilidade e escalabilidade, bem como colaborações entre *hardware* e *software*. Deste modo, a utilização de FPGAs é um meio poderoso e versátil para acelerar a pesquisa arquitectónica e desenvolvimento de *software*, pois a simulação tradicional de sistemas multi-processador requer um tempo de simulação demasiado elevado. Arquitecturas deste tipo anteriores, como as referidas anteriormente, apresentam vários tipos de problemas, nomeadamente devido a sobrecarga provocada pela comunicação entre componentes simulados distribuídos. Nesta arquitectura proposta em [10], tem como objectivo reduzir a complexidade do sistema através da (virtualização) implementação de vários processadores numa única FPGA e através de uma técnica de simulação híbrida, implementar na FPGA só as tarefas mais repetitivas (paralelas e recorrentes recursivas), de maneira a que o simulador de *software* mantenha a abstracção do sistema completo. Por virtualização, o número de máquinas pode ser escolhido judiciosamente, consoante o necessário, para entregar o desempenho de simulação necessário.

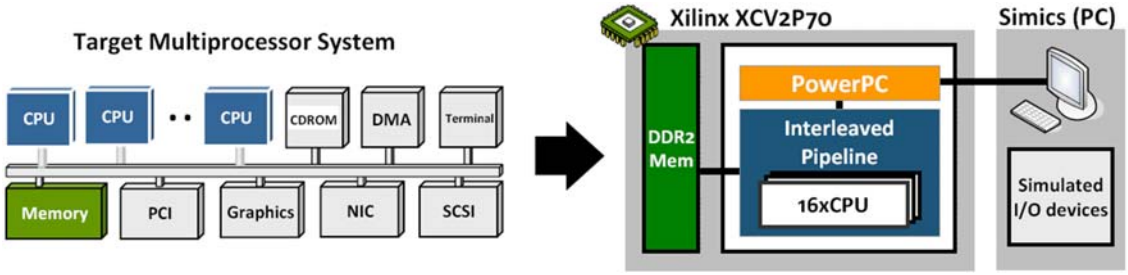


Figura 2.16: Modelo do sistema proposto em [10].

Nesta figura é possível observar o diagrama de blocos da arquitectura proposta para simular um sistema multiprocessador simétrico composto por 16 processadores, instalado em uma única FPGA Virtex-II da Xilinx. Com esta arquitectura, a simulação obtida mostrou um significativo aumento de velocidade, 39 vezes em média (49 vezes no melhor caso) mais rápida em relação com a simulação apenas em *software* através de um conjunto de aplicações, confirmando assim as vantagens deste tipo de arquitectura para a simulação de sistemas de multi-core.

Os resultados obtidos nestes estudos mostram assim, o potencial das FPGAs e o motivo pelo qual estas cada vez mais são utilizadas em várias áreas em detrimento de outras plataformas digitais.

2.5 Máquinas de estados finitos

2.5.1 Introdução

Uma máquinas de estados finitos (FSM), é um meio bastante útil para descrever sistemas em *hardware*, que transitam entre um número finito de estados. Ao contrário de um simples circuito sequencial, em que as transições de estado seguem um padrão simples e repetitivo, numa FSM estas transições dependem do estado actual e de entradas externas. Uma FSM é conhecida como uma máquina de *Moore* quando as saídas dependem só do estado actual, e é conhecida como uma máquina de *Mealy* se as saídas são uma função do estado actual e das entradas externas.

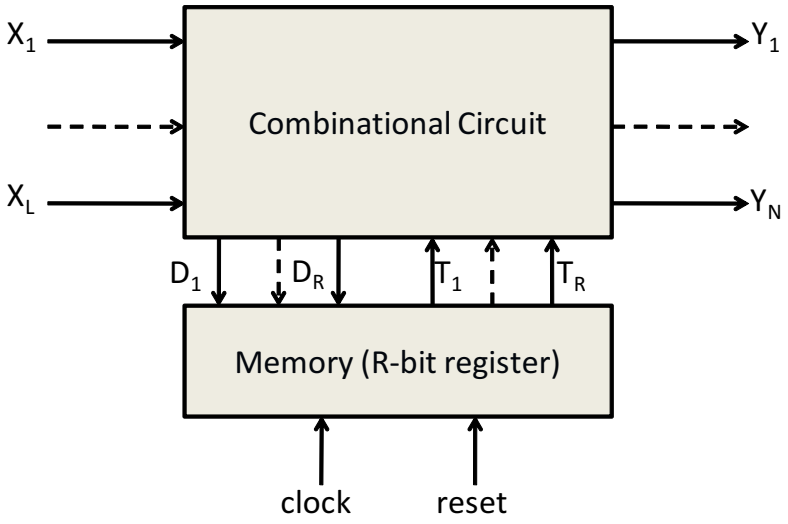


Figura 2.17: FSM [11].

Como se pode observar pela figura anterior, uma FSM é composta estruturalmente

[26]por um circuito combinatório que calcula as transições entre os estados e as saídas da máquina e duma memória que contém o estado actual. Assim, uma FSM pode ser definida por um sêxtuplo (S, X, Y, ϕ, ψ, s1), onde:

- S={s1,...,sM} é o conjunto finito de estados;
- X={x1,...,xL} é o conjunto de entradas;
- Y={y1,...,yN} é o conjunto de saídas;
- ϕ é a função de transição entre estados;
- ψ é a função de saída;
- s1 $\in$ S é o estado inicial.

A principal aplicação prática de uma FSM é funcionar como um controlador de um sistema digital que é sensível a entradas externas e activa sinais próprios que controlam um conjunto de dados que normalmente é composto por circuitos sequenciais. Deste modo, um sistema destes também é conhecido como uma FSMD (máquina estados finitos com *datapath*) quando combina uma FSM (*controlpath*), com simples circuitos sequenciais (*datapath*)[27].

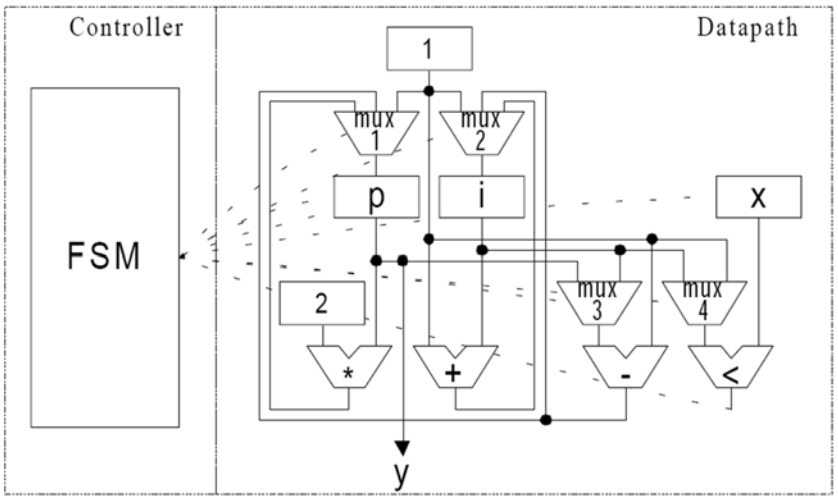


Figura 2.18: Exemplo de uma FSMD.

Para a implementação de uma FSM(D) pode ser usado um dispositivo lógico programável, que contenha portas lógicas e *flip-flops*, ou seja, que disponha de *hardware* necessário á construção de simples circuitos sequencias, registos que permitam guardar os valores das variáveis, e blocos de lógica combinatória para a determinação das transições entre os vários estados e saídas da máquina de estados finitos. Deste modo, uma

FPGA torna-se num dispositivo indicado para este tipo de sistemas, pois esta contém todos estes requisitos físicos necessários para a implementação em *hardware* de uma FSMD.

Estas também podem ser aplicadas em muitos casos em substituição de ciclos *for* combinatórios, pois muitas vezes estes ciclos não são sintetizáveis em algumas ferramentas de *software*. Esta substituição também pode ser bastante útil, para operações não criticas em relação ao tempo, pois com uma FSM em vez de um ciclo *for* é normalmente possível poupar uma grande quantidade de recursos lógicos, uma vez que a medida que o ciclo aumenta, os recursos lógicos ocupados pela FSM são sempre os mesmos.

Estas simples máquinas de estados, apresentadas em cima, tiveram um papel fundamental para o desenvolvimento e evolução dos sistemas de controlo digital, pois com base nestas foram desenvolvidas máquinas de estados avançadas, que fornecem um basto leque de possibilidades no controlo digital e na programação de *hardware*. Estas FSMs avançadas tornam assim possível, algoritmos modulares, hierárquicos, recursivos, paralelos e auto-reconfiguráveis serem implementados e mapeados em *hardware*.

Uma dessas possibilidades, proposta em [28], são as máquinas de estado finitas que podem ser reconfiguradas durante sua execução. Com a ajuda dos dispositivos lógicos programáveis como as FPGAs, este modelo pode torna-se importante na implementação de *hardware* reconfigurável. Aqui é mostrada uma solução da implementação desta reconfiguração em *hardware* através da transição de uma determinada FSM numa nova FSM. Uma FSM é auto - reconfigurável, se a reconfiguração da sua função de saída e transição for iniciada por ela própria e não por comandos externos. Isto é feito utilizando uma RAM na construção do circuito combinatório da FSM [26].Como este tipo de FSMs não faz parte do âmbito deste projecto, não será aprofundado aqui. Para informações mais detalhadas pode ser consultados os documentos [26, 28].

2.5.2 Máquinas de estados finitos hierárquicas

Um tipo de FSMs muito importante no controlo de digital, são as máquinas de estados finitos hierárquicas (HFSM). Em [29] podemos encontrar uma descrição comportamental e uma implementação deste tipo de FSMs, bem como uma análise ao seu uso prático para unidades de controlo. Com este tipo de máquinas, é possível a decomposição de algoritmos complexos, em algoritmos mais pequenos e simples através de uma decomposição de cima para baixo, com um maior nível de abstracção. Este tipo de abordagem dá-nos a possibilidade de fazer chamadas recursivas em hardware, fornecendo assim uma grande flexibilidade às unidades de controlo. Estas são compostas por módulos

que podem ser outras HFSMs ou simples FSMs, um pouco a semelhança com o que acontece com os procedimentos nos programas de *software*.

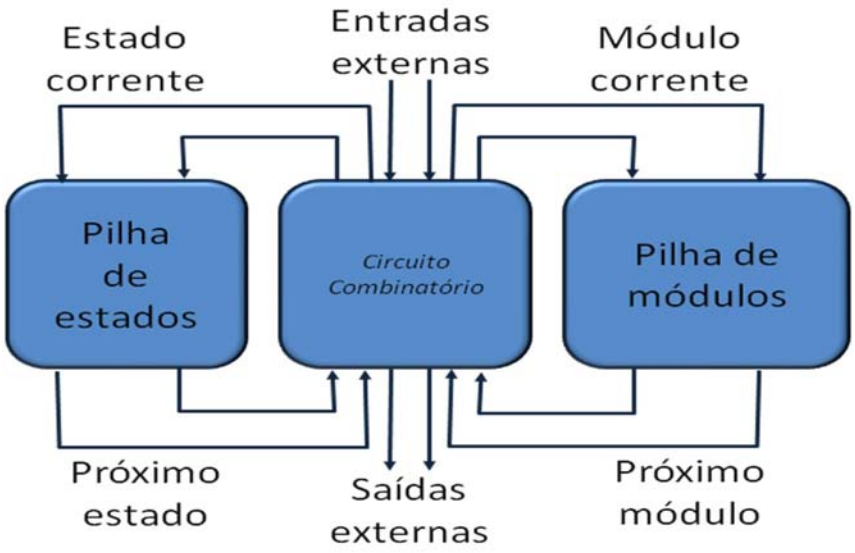


Figura 2.19: HFSM[11].

Uma HFSM como se pode observar na figura acima, contém duas pilhas, uma para guardar os estados e outra para guardar os módulos. Aqui cada módulo pode ser visto, como uma simples FSM, onde o seu estado actual e guardado na respectiva posição da pilha de estados, ou seja, na posição em que se encontra o módulo na pilha de módulos. Um elemento muito importante neste tipo de arquitectura é o ponteiro das pilhas que é comum a ambas as pilhas, pois a cada módulo activado corresponde um respectivo estado activo. Cada módulo executa uma determinada função, abstraindo-se assim das funções de outros módulos, fornecendo assim um nível de abstracção como era pretendido. Deste modo, como cada módulo funcionam individualmente, cada um tem um código único, e assim o mesmo código de estado podem ser repetidos em módulos diferentes. O circuito de combinatório interage com as pilhas, em tudo semelhante ao que acontece com a memória que contém o estado actual numa simples FSM descrita em cima. Este é responsável pelas transições entre os estados do módulo que está activo, pela invocação de novos módulos, bem como o retorno a módulos hierarquicamente acima e também fornece as saídas externas da máquina. Numa transição entre os estados do mesmo módulo, ou seja, não hierárquica, apenas e efectuada uma mudança no código na posição apontada pelo ponteiro na pilha de estados. Já numa chamada hierárquica, o ponteiro das pilhas é incrementado e são alterados os estados de ambas as pilhas nas posições respectivas, de maneira que a pilha dos módulos guarde o código do novo módulo activado e na pilha dos estados será adicionado o estado inicial

desse módulo, normalmente o a0=0...0. Estas alterações nas pilhas são efectuadas na nova posição do ponteiro, evitando assim perder informação do módulo activo anteriormente, necessário ao retorno hierárquico, onde é decrementado apenas o ponteiro sem alterações nas pilhas, continuando assim a execução da máquina no respectivo módulo anterior, no estado também guardado anteriormente para esse efeito. A execução da HFSM acaba quando o ponteiro das pilhas é igual a zero e o algoritmo chega ao último estado do módulo activo (módulo mais acima na hierarquia).

```
process(clk,rst)
begin
if rst = '1' then
        stack_counter <= 0;
        FSM_stack(stack_counter) <= a0;
        M_stack(stack_counter) <= z0;
        error <= '0';
  elsif rising_edge(clk) then
        if inc = '1' then
                if stack_counter = stack_size then error <= '1';
                else
                        stack_counter <= stack_counter + 1;
                        FSM_stack(stack_counter+1) <= NM_FS;
                        FSM_stack(stack_counter) <= NS;
                        M_stack(stack_counter+1) <= NM;
                end if;
        elsif dec = '1' then
                  stack_counter <= stack_counter - unwinding;
        else      FSM_stack(stack_counter) <= NS;
        end if;
end if;
end process;
```

Figura 2.20: Processo em VHDL para o controlo das pilhas.

Os componentes funcionais primários do algoritmo do controlo podem ser usados de novo em várias aplicações futuras, como por exemplo o processo da figura 2.20. Este tipo de FSM baseada em hierarquia é muito útil para várias aplicações práticas, pois permite o controlo de uma grande gama de dispositivos, oferecendo a capacidade de chamadas recursivas e uma grande flexibilidade. No entanto, como se pode verificar em [29], os melhores resultados obtidos foram para a implementação de algoritmos recursivos. Assim uma HFSM é uma solução bastante eficiente para implementação de algoritmos modulares, hierárquicos e recursivos em hardware. Estas propriedades tornam-se importantes, pois assim é possível centralizar-nos no que as operações fazem em vez de como as implementar [29].

2.5.3 Implementação de chamadas recursivas em hardware

Técnicas de implementação de chamadas recursivas em hardware, tornam-se bastante importantes á medida que as FPGAs começam a mostrar melhor desempenho que microprocessadores em muitas áreas de aplicação por causa da melhoria drástica do seu tamanho e velocidade. Deste modo, já existem muitos estudos em como tirar partido das vantagens inerentes as FPGAs, envolvendo estas directamente com microprocessadores, de maneira a estas executarem algoritmos escritos em linguagens de programação, e permitir assim um aumento do desempenho do sistema. Neste contexto, nos algoritmos escritos em linguagens de programação de *software*, as chamadas recursivas e os ciclos repetitivos são as partes dos algoritmos que ocupam mais tempo, sendo assim as principais responsáveis pelo atraso do sistema. Deste modo, técnicas de implementação de chamadas recursivas em hardware tornam-se vitais para um melhor desempenho do sistema. Não existe nenhum modo universal para a implementação de chamadas recursivas em *hardware*, no entanto é possível encontrar na literatura várias soluções particulares para a implementação em *hardware* desta técnica poderosa e fundamental dentro do *software*.

Método 1

No método proposto em [30], onde as recursividade foi implementada utilizando pilhas, são apresentados resultados muito bons, uma vez que as chamadas recursivas e ciclos repetitivos implementados em *hardware* reconfigurável apresentam um melhor desempenho desde 4.1 - 6.7 vezes mais rápido quando comparado com implementação em *software* numa tradicional estação de trabalho.

Método 2

Outro método é apresentado em [31], onde a implementação da recursividade não utiliza pilhas, de maneira a tentar tirar maior proveito dos recursos paralelos presentes num dispositivo reconfigurável como uma FPGA. Deste modo, este método decompõe a chamada recursiva em duas funções, a pré-recursiva e a pós-recursiva. Assim distingue as duas partes em funções independentes e coloca-as no *pipeline* (linha de execução). Nesta implementação é utilizado o conceito de reconfiguração em tempo real (RTR) de forma a extrair assim para dentro da FPGA o "paralelismo" da chamada recursiva. Contudo, isto impôs novas restrições para no projecto do *pipeline* quer em área como em tempo como se pode ver com maior detalhe em [31]. Devido a área que este método pode consumir, o *pipeline* está sujeito ao constrangimento natural de área que é imposto

pelo tamanho da FPGA usada para o efeito. Assim se a recursão for suficientemente longa, o *pipeline* não será suficiente para cobrir toda a recursão devido à falta de espaço na FPGA. Em [31] também são apresentadas técnicas de modo a reduzir o impacto de RTR no desempenho, bem como para lidar com as limitações impostas pelos recursos naturalmente finitos de hardware. Os resultados apresentados por este método, demonstram uma melhoria significativa no desempenho quando comparado por uma implementação baseada numa pilha. Esta diferença de desempenho aumenta á medida que aumenta a quantidade de dados a processar, no entanto este método não é aplicável a funções recursivas mais genéricas (com taxa de crescimento maior que 1).

Método 3

Em [32] são comparados estes tipos implementações de chamadas recursivas em hardware, e também é proposto por sua vez um novo método que utiliza uma aproximação orientada aos dados, onde são os dados armazenados na pilha de dados a decidir qual será o próximo estado. Deste modo, esta implementação apresenta vantagens importantes, pois simplifica a lógica e o *hardware* necessário para as transições entre os estados e nas chamadas recursivas, apresentando assim um design mais compacto. Esta arquitectura apresenta um desempenho superior em relação á implementação dos algoritmos em *software* (até 3 vezes mais rápido), e uma menor utilização de recursos físicos como era esperado, em relação a outras implementações em *hardware*, descritas anteriormente. É demonstrado que esta técnica pode ser eficazmente implementada de um modo geral nas FPGAs, dependendo apenas do design, que é constrangimento comum a todas implementações já descritas anteriormente.

Método 4

O método utilizado neste trabalho para a implementação de chamadas recursivas em hardware foi baseado em [33]. Esta implementação utiliza a HFSM descrita anteriormente, introduzindo todas as vantagens descritas anteriormente inerentes a hierarquia, permitindo assim a implementação de um algoritmo recursivo. No entanto, para melhorar o desempenho desta máquina nas chamadas recursivas, foram feitas ligeiras alterações á HFSM inicial de modo a diminuir assim o tempo necessário as invocações recursivas. Este modelo em[33] foi chamado como RHFSM (*recursive* HFSM). O funcionamento desta máquina é em tudo semelhante á HFSM, sendo que a alteração principal consiste na introdução de um novo sinal de "*return_flag*" que permite optimizar o controlo das chamadas hierárquicas e consequentemente das chamadas recursivas, simplificando assim o desenho e implementação de algoritmos recursivos em *hardware*,

fornecendo a flexibilidade de um projecto em *software*. Com esta máquina a recursividade pode ser implementada em *hardware* muito eficientemente, contudo, devido ao uso de várias pilhas, para algoritmos recursivos longos esta técnica pode consumir muitos recursos físicos, para o armazenamento dos vários módulos e estados respectivos. Deste modo, para resolver este problema as pilhas devem ser implementadas em blocos de memória embutidos (*Block Rams*) disponíveis actualmente em qualquer FPGA, aumentando assim a eficiência desta técnica. Todos os detalhes e processos em VHDL sintetizáveis, necessários para a construção de uma RHFSM estão disponíveis em [33] , sendo estes facilmente escaláveis a todo o tipo de aplicações recursivas. Esta máquina foi testada em duas aplicações distintas, na ordenação recursiva de uma árvore binária e na compressão de dados.

Em [34] através de implementações baseadas em FPGAs são comparados algoritmos recursivos e repetitivos (interactivos), para a resolução de vários tipos de problemas. Aqui são apresentados critérios para a escolha de implementações recursivas ou repetitivas dependendo de cada problema particular. Foi demonstrado que para problemas de procura binária, os algoritmos recursivos são melhores, quando comparados com algoritmos repetitivas. Pois embora a implementação repetitiva use menos recursos físicos, esta apresenta um maior tempo de execução que a implementação recursiva, ao contrário do que acontece em *software* em que o algoritmo repetitivo apresenta um desempenho ligeiramente melhor. Esta diferença de desempenhos, deve-se ao facto, como já vimos anteriormente, que algoritmos recursivos podem ser implementado em *hardware* muito mais eficazmente que em *software*. Com a utilização de blocos embutidos de memória (*Block Rams*) nos algoritmos recursivos, os recursos físicos utilizados por ambas implementações tornam se praticamente os mesmos. Apesar da escolha ser sempre um pouco subjectiva, em aplicações de maior complexidade, uma especificação hierárquica torna-se importante de modo a simplificar e clarificar o algoritmo, e deste modo, chamadas recursivas também podem ser feitas sem qualquer *hardware* adicional, bem como o mesmo módulo ser reutilizado e permitir assim redução do tempo de *design* e de recursos físicos.

2.5.4 Máquina de estados finitos hierárquica paralela

Por fim, em [35] é apresentado modelo de uma FSM avançada que permite vários algoritmos serem executados em paralelo, através de uma máquina de estados finitos hierárquica Paralela (PHFSM). Esta PHFSM utiliza toda a metodologia descrita anteriormente para a HFSM, no entanto, com esta é possível ter mais ter mais do que

um módulo activo ao mesmo tempo, ou seja, estes podem ser executados em paralelo. Assim dentro da mesma FSM é possível combinar capacidades hierárquicas e paralelas. Os detalhes deste modelo PHFSM podem se observar na próxima figura:

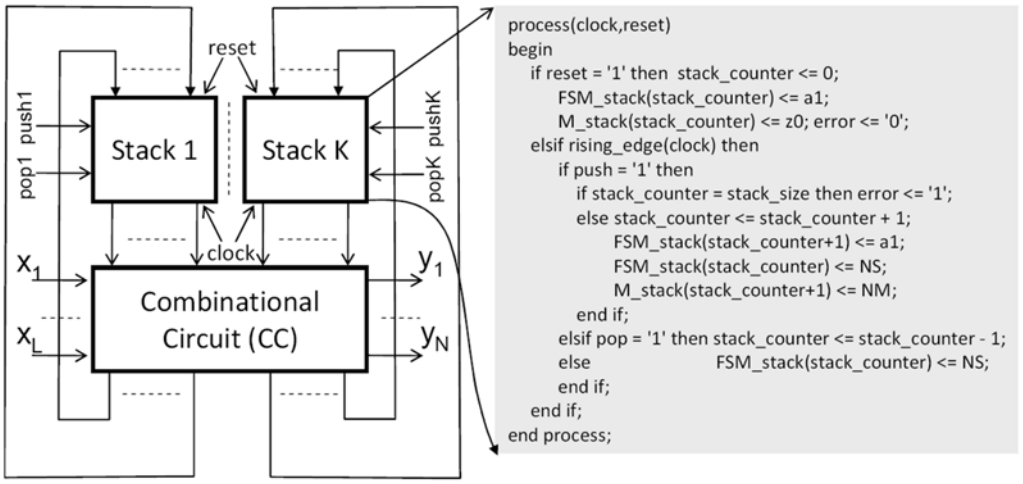


Figura 2.21: PHFSM e *template* do código em VHDL[11].

Para o seu funcionamento são utilizadas pilhas múltiplas que interagem com um circuito combinatório. Consoante o número de pilhas (k), serão possíveis k módulos a serem executados em paralelo. A cada pilha está associada um sinal *push* e um sinal *pop*. Estas pilhas apresentam uma estrutura, em tudo semelhante ao que acontece na HFSM anterior, como se pode verificar, comparando os dois templates das figuras 2.20 e 2.21. Neste caso será necessário k processos como este, um para cada pilha. Deste modo, tal como sucede na HFSM, cada pilha da figura 2.21 (*stack* k) contém duas partes, uma para os módulos e outra para os estados, ambas partilham o mesmo ponteiro que é incrementado e decrementado pelos sinais de *push* e *pop*, respectivamente. No entanto, e necessário ter em atenção algumas regras na utilização desta máquina PHFSM, para um funcionamento correcto do sistema:

- Não é possível chamadas paralelas num módulo recursivo;
- Não deve ser activado o módulo pai num módulo que está a ser executado em paralelo;

Se dois ou mais módulos são invocados ao mesmo tempo num dado modulo, este só pode continuar a sua execução quando todos os módulos chamados por este, estiverem terminados, ou seja, se um dos módulos que estão em paralelo acabar a sua execução primeiro que o outro, o modulo pai tem de continuar suspenso até o outro acabar. O código completo e sintetizável em VHDL de um exemplo prático desta maquina PHFSM

está disponível *online* em [36]. Este código pode ser utilizado como um template para outras aplicações. Este código também já contém todas as alterações necessárias para evitar invocações repetidas, de acordo com as sugestões em [10]. Vários algoritmos foram implementados usando PHFSMs, bem como um simples controlador embutido para a demonstração na prática, em FPGAs de baixo custo. Com isto, obtiveram-se resultados bastante satisfatórios no que diz respeito aos recursos físicos utilizados, mesmo não utilizando blocos de memória embutida, o que pode melhorar ainda esses resultados. Deste modo, estes resultados [14] mostram que a implementação desta máquina PHFSM é benéfica e possível de usar na prática, permitindo assim a utilização desta na resolução de problemas do dia-a-dia, que envolvam algoritmos hierárquicos e paralelos uma vez que não consome muitos recursos físicos.

Capítulo 3

Ferramentas de desenvolvimento

Sumário

Neste capítulo é feita uma introdução desde os primeiros dispositivos lógicos programáveis até as FPGAs. São apresentadas as características fundamentais deste dispositivo, desde a sua estrutura, vantagens, aplicações até a sua evolução, e como em conjunto com as LDH e os blocos IP disponíveis revolucionaram os sistemas digitais. Depois disto, também são apresentadas as principais etapas no desenvolvimento de sistemas baseados em FPGA e as ferramentas de suporte que foram utilizadas neste trabalho (ISE e Modelsim). Neste capítulo são ainda apresentadas as placas de desenvolvimento utilizadas neste trabalho (Celoxica RC10 e Nexys 2). Finalmente são descritas as características da ligação sem fios pretendida para interacção remota e deste modo é apresentado o *Transceiver* RF escolhido para esse efeito, o CC1101 da Texas Instruments.

3.1 Introdução aos dispositivos lógicos programáveis

Um dispositivo lógico programável (*Programmable logic device* - PLD) é um circuito integrado utilizado para construir circuitos digitais, que podem ser configurados, a fim de implementar uma função desejada pelo utilizador, ao contrário de uma porta lógica, que tem uma função fixa definida no seu fabrico. Assim estes dispositivos lógicos programáveis caracterizam-se por uma grande flexibilidade de utilização, bem como de custos e tempo de desenvolvimento reduzidos. Dispositivos lógicos programáveis estão disponíveis numa basta gama de arquitecturas e bem como de tamanhos. Estes podem ser divididos em três tipos de capacidade, características e custo crescente com a seguinte ordem [37, 38, 12]:

- SPLD: Dispositivos lógicos programáveis simples;
- CPLD: Dispositivos lógicos programáveis complexos;
- FPGAs: *Field Programmable Gate Array*;

Os dispositivos dentro do tipo SPLD contém tipicamente no máximo até 600 gates [37], já os do tipo CPLD costumam ter até 10,000 gates e por fim a FPGA pode conter até milhões de gates. Devido a este facto, CPLD e FPGA fazem parte do grupo de dispositivos lógicos programáveis de alta capacidade (HCPLD).

3.1.1 SPLDs

No grupo dos dispositivos SPLD podemos ter os seguintes tipos:

- PROM (Programmable ROM);
- PLA (Programmable Logic Array);
- PAL (Programmable Array Logic).

O primeiro tipo de dispositivo lógico programável utilizado, foram as memórias ROM (*read-only memory*). Esta memória não volátil pode ser carregada com informação e assim programada com uma função definida pelo utilizador. Deste modo, para uma ROM com m de entradas (Linhas de endereço) e n de saídas (Linhas de dados), esta torna-se equivalente a n de circuitos lógicos independentes, cada qual gerado em função das m entradas, tornando este dispositivo lógico disponível para aplicações mais abrangentes. As PROM (Rom programável), podem ser de dois tipos: EPROM (*Erasable* PROM) e EEPROM (*Electronic Erasable* PROM). A mais usada normalmente é a EEPROM que permite ao utilizador apagar o conteúdo da memória e reprogramá-la muitas vezes. No entanto, as PROM apresentam algumas desvantagens:

- São mais lentas que os circuitos lógicos dedicados;
- Consomem mais energia;
- Ineficiência na utilização do espaço e consequentemente da sua capacidade devido a sua arquitectura.

Figura 3.1: Exemplo de uma PROM.

O próximo passo na evolução de lógica programável foi o desenvolvimento das PLAs e PALs em meados dos anos 70, desenvolvidos especialmente para a implementação de circuitos lógicos. Estes dispositivos são compostos geralmente por um conjunto de portas lógicas AND programáveis seguidas por outro conjunto de portas lógicas OR programáveis para o caso das PLAs e fixas para as PALs. Esta arquitectura é mais flexível, no entanto apresenta atrasos de propagação que torna os dispositivos relativamente lentos.

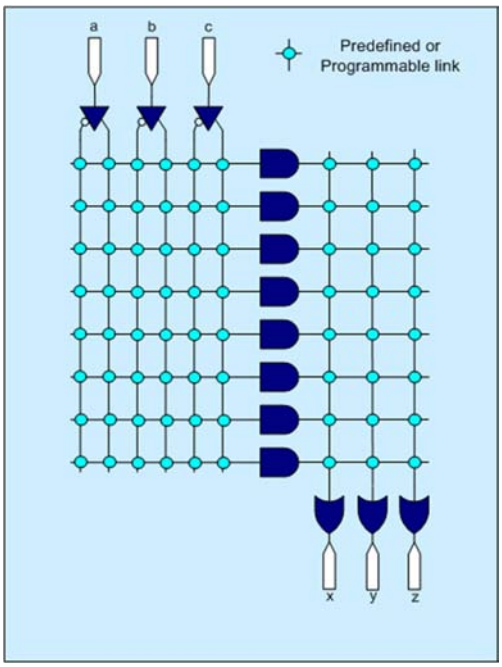


Figura 3.2: Estrutura das PLAs e PALs[12].

Os PLAs tiveram sucesso limitado devido ao seu preço de fabrico, para além disso, os atrasos de propagação são significativos. Os PALs por sua vez asseguram o desempenho elevado, uma vez que estes são mais rápidos, pois só contém um conjunto de portas (AND) programável. No entanto, isto limita o conjunto de condições de produto que podem ser geradas. Para compensar esta menor flexibilidade, são produzidas várias versões deste dispositivo com diferentes números de entradas, saídas, e vários tamanhos[26]. Normalmente, os PALs contêm flip-flops ligados às saídas de portas lógicas OR o que permite realizar circuitos sequenciais. Deste modo, estes PLDs mostram um caminho claro para o desenvolvimento das FPGAs.

3.1.2 CPLDs (Complex Programmable Logic Devices)

Dispositivos de lógica programáveis complexos (CPLDs) são dispositivos com maior capacidade que a dos SPLDs, uma vez que apresentam uma maior densidade lógica,

permitindo assim estes serem aplicados numa maior gama de aplicações. Estes são implementados com o uso de alguns blocos de PLD já referidos acima em um único dispositivo com ligações programáveis entre estes blocos.

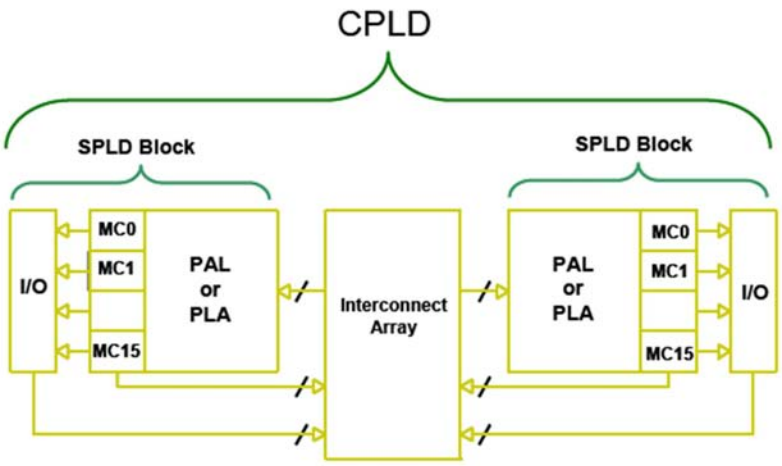


Figura 3.3: Estrutura de um CPLD [13].

Geralmente, CPLDs apresentam atrasos mais curtos e mais previsíveis que outros dispositivos lógicos programáveis, sendo relativamente económicos quer ao nível do consumo de energia, quer nos custos de desenvolvimento [12]. Deste modo estes são frequentemente usados em aplicações portáteis alimentadas por bateria. Uma grande diferença entre um CPLD e uma FPGA é a tecnologia de base ser memória não volátil no caso dos CPLDs. Esta diferença é cada vez menos importante devido ao facto dos últimos produtos com FPGA também incluem memória não volátil de configuração embutida. Esta característica faz do CPLD um dispositivo de escolha em sistemas digitais modernos para executar funções '*boot loader*' antes de passar o controlo para outros dispositivos que não têm esta capacidade. Um exemplo de uma aplicação é quando um CPLD é usado para carregar dados de configuração para uma FPGA.

3.2 FPGAs

FPGAs (Field Programmable Gate Arrays) são os dispositivos lógicos programáveis de maior capacidade disponíveis hoje em dia. As FPGAs são assim a ultima evolução de PLDs, podendo estas serem programadas depois de serem fabricadas. Deste modo, uma FPGA não está restringida a uma função qualquer de *hardware* predeterminada no seu fabrico, podendo esta ser programada em função das características do produto e aplicação desejada. Pode adaptar-se assim a novos padrões e funções através da

reconfiguração do *hardware* para aplicações específicas até mesmo depois de o produto ser instalado.

Figura 3.4: FPGAs.

Assim, em vez de um ASIC (*application-specific integrated circuit*) que é um dispositivo construído para uma função particular, pode ser utilizada uma FPGA, mas com a vantagem de se puder actualizar a funcionalidade para aplicações diferentes, uma vez que se trata de um *hardware* reconfigurável, em que as suas funcionalidades são definidas pelos utilizadores e não pelos fabricantes. Deste forma estas podem ser reutilizáveis em aplicações diferentes, possuindo assim um maior ciclo de vida.

Estas contêm um grande número de blocos lógicos configuráveis contidos em um único circuito integrado de maneira a suportar a implementação de circuitos lógicos de complexidade elevada. Quando um circuito lógico é implementado em uma FPGA, os blocos lógicos são programados para realizar as funções necessárias, e as ligações são feitas de forma a realizar a interconexão necessária entre os blocos lógicos. Uma FPGA é composta basicamente por três tipos de componentes como se pode ver na figura 3.5: blocos de entrada e saída (IOB), blocos lógicos configuráveis (CLB) e as interconexões programáveis (PI).

3.2.1 Blocos lógicos configuráveis (CLB)

O bloco lógico configurável é a unidade básica de uma FPGA, números exactos e características variam de dispositivo a dispositivo. No geral, esta unidade consiste em uma matriz configurável de 4 ou 6 entradas também designadas por *lookup tables* (LUTs), registos, circuitos multiplexadores, e flip-flops [39]. Esta matriz é altamente flexível e pode ser configurada como por exemplo como uma memória RAM (memória distribuída) ou como lógica combinatória. Note que as saídas das LUTs podem ser usadas

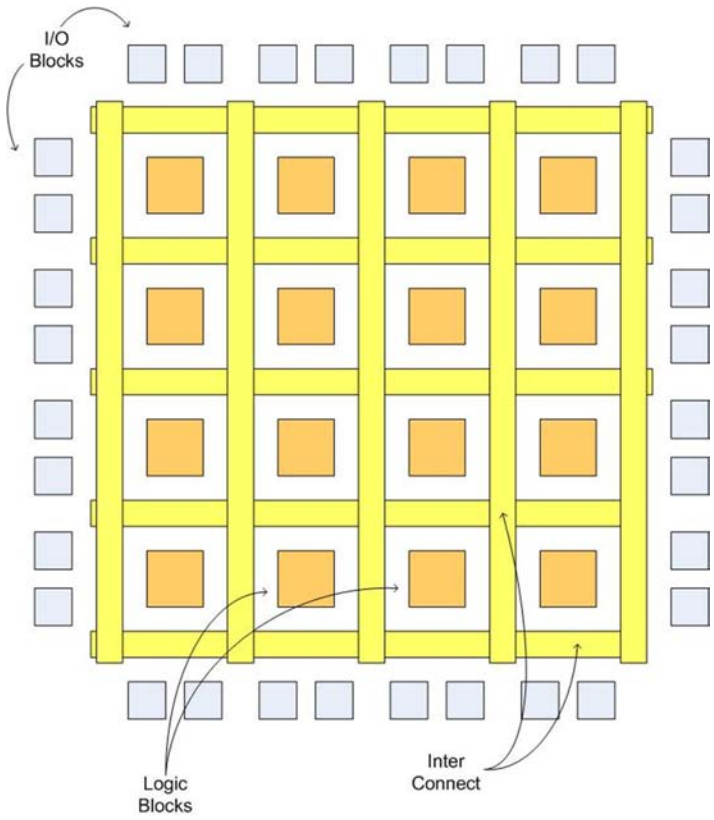


Figura 3.5: Estrutura básica de uma FPGA.

directamente ou serem armazenadas nos flip-flops. Este último caso pode ser usado para implementar circuitos sequenciais. No caso da FPGA utilizada neste trabalho, Spartan-3 da Xilinx, a matriz configurável tem 4 entradas e assim sendo pode ser configurada como uma memória RAM 16-por-1[27].

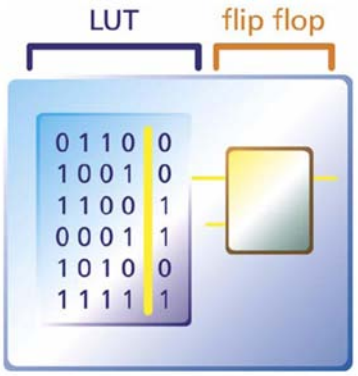


Figura 3.6: Estrutura básica de um bloco lógico SRAM[13].

Existem dois tipos básicos de FPGAs[13], estes dois tipos de FPGAs diferem na implementação dos blocos lógicos e no mecanismo que faz conexões entre estes. O tipo

dominante de FPGA (que foi utilizado neste trabalho) é a baseada em SRAM e pode ser reprogramado o número de vezes que o utilizador o entender, e sempre que esta a FPGA é ligada uma vez que se trata de uma memória volátil como já vimos atrás.

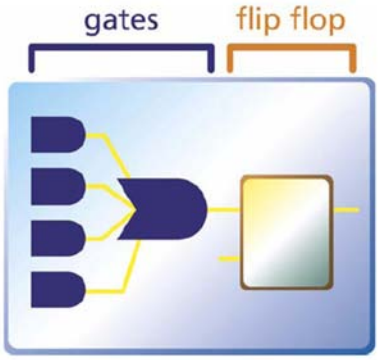


Figura 3.7: Estrutura básica de um bloco lógico OTP[13].

O outro tipo de FPGAs são as OTP (*One Time Programmable*), estas usam conexões permanentes no circuito integrado. Deste modo os sistemas com este tipo de FPGAs não requerem nenhuma PROM para a sua programação. Contudo, perde a grande vantagem das FPGAs, ou seja, a sua reconfigurabilidade. Esta lógica é em tudo semelhante as PLDs, com *gates* e *flip-flops* dedicados.

3.2.2 Blocos de entrada e saída (IOB)

As FPGAs de hoje em dia apresentam uma grande flexibilidade na sua utilização, fornecendo suporte para dúzias de padrões de I/O fornecendo assim a interface ideal para diferentes tipos de sistema. Estes fazem a interface entre a lógica interna da FPGA com os pinos de encapsulamento e podem ser configurados como só de entrada ou saída, ou ainda como bidireccionais com capacidade *tri-state*.

3.2.3 Interconexões programáveis (PI)

Enquanto os CLBs fornecem a capacidade de lógica, as interconexões programáveis fazem o encaminhamento dos sinais entre CLBs e com os blocos I/O. As ferramentas de *software* de design abstrai do utilizador tarefa de encaminhamento a menos que este especifique o contrário, reduzindo assim significativamente a complexidade do projecto. Deste modo a ferramenta de *software* escolhida tem um papel fundamental na gestão e optimização dos recursos lógicos da FPGA utilizada.

3.2.4 Macro blocos embutidos em FPGA

As FPGAs de hoje evoluíram muito para além das capacidades básicas apresentadas nas suas antecessoras e contém e diferentes tipos de blocos embutidos com funções especificas como blocos de memória RAM, multiplicadores combinatórios, circuitos de geração de relógio (DCM), blocos de processamento digital de sinal (DSP), *transceivers* de alta velocidade, entre outros. Estes blocos são desenhados e fabricados ao nível do transístor e as suas funcionalidades complementam os blocos lógicos configuráveis. Dispositivos de FPGA avançados podem conter um ou mais *cores* de processador pré-fabricados como já vimos anteriormente.

Dentro destes macro blocos, os mais comuns e importantes são os blocos de memória embutida (*Block RAMs*) que estão disponíveis na maioria das FPGAs actuais, sendo bastante útil aos projectos implementados em FPGA, pois assim não será necessário utilizar recursos lógicos como memória, permitindo deste modo uma poupança enorme de recursos lógicos das respectivas FPGAs. Com estes blocos, as FPGAs podem conter até 10 Mbits de memória embutida, que suportam operações em dois portos diferentes na mesma memória. Administração de relógio digital (DCM) é fornecida pela maioria das FPGAs e eliminou quase todos os problemas que os utilizadores tiveram que enfrentar no passado para projectar sinais globais em FPGAs. Com este bloco o utilizador pode controlar a frequência e a fase do sinal do relógio, sem problemas adicionais, reduzindo assim o tempo de projecto.

3.2.5 Vantagens

Desde a primeira FPGA disponível comercialmente pela empresa Xilinx Inc, em 1985, tem-se assistido a um aumento enorme da sua utilização, devido principalmente as seguintes vantagens que as FPGAs oferecem [11]:

- Reconfiguravilidade - esta é a sua principal vantagem, pois permite a reutilização do mesmo circuito integrado, ou seja, este pode ser utilizado para implementação de sistemas diferentes;
- Eficiência - tempo de desenvolvimento do projecto reduzido quando comparando com outras tecnologias;
- Paralelismo - possibilidade de executar operações arbitrárias em paralelo;
- Virtualização - algumas partes do sistema podem ser alteradas durante execução, podem ser reconfiguradas parcialmente;

- Alta densidade - milhões de portas lógicas programáveis.

Deste modo, as FPGAs tem vindo a ganhar espaço em relação a outras tecnologias como já vimos anteriormente, como por exemplo aos ASIC ou ASSP, uma vez que estas plataformas de *hardware* são muito específicas e inflexíveis, pois as suas funções não podem ser alteradas depois do seu fabrico, como no caso das FPGAs. Estas apresentam assim grande flexibilidade, ao permitir aos projectistas a implementarem algoritmos directamente no silício, e deste modo é possível melhorar os produtos do consumidor final sem este ter de comprar um dispositivo novo. Ainda assim, a escolha entre ASICs e FPGAs na implementação de uma determinada solução deve ser feita cuidadosamente depois de avaliar as vantagens de cada tecnologia, pois uma implementação em ASIC também apresenta a suas vantagens, uma vez que neste caso o circuito é optimizado para uma função específica proporcionando assim na maioria dos casos menores consumos e maiores velocidades, bem como oferece custos unitários inferiores, para grandes volumes de produção.

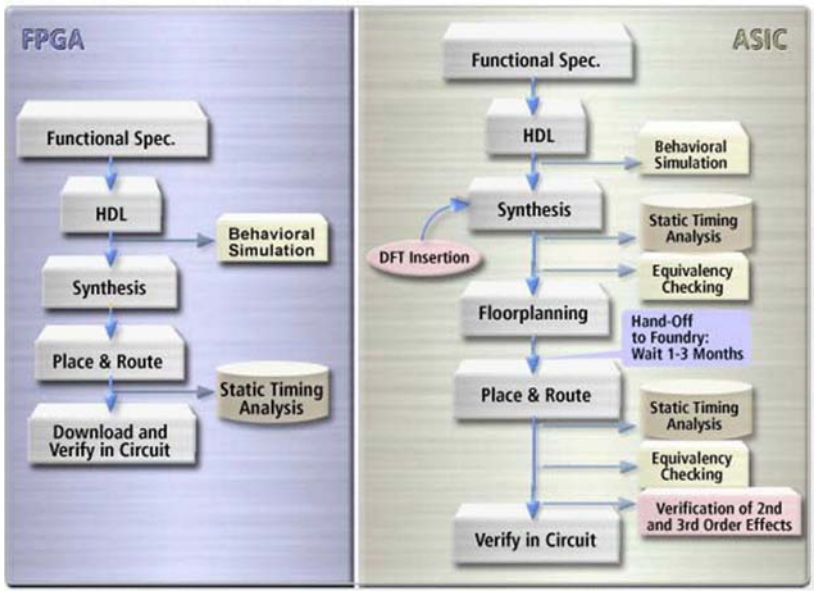


Figura 3.8: Comparação do fluxo de desenvolvimento entre FPGA e ASIC [14].

Como se pode observar pela figura 3.8, no fluxo de projecto de uma FPGA são eliminadas etapas de longa duração (*floorplanning, timing analysis, mask*),uma vez que neste caso a lógica já é sintetizada para ser implementada numa determinada FPGA já conhecida, simplificando e reduzindo assim o tempo de desenvolvimento do projecto. No entanto estas etapas podem ser consideradas, se necessário, quando os utilizadores pretendem maximizar o desempenho dos projectos mais exigentes. Deste

modo, uma implementação em FPGA permite implementar e testar sistemas de uma forma bem mais rápida e numa basta gama de aplicações.

3.2.6 Aplicações

Deste modo, devido à sua natureza programável e suas vantagens já enumeradas em cima, as FPGAs são assim a solução ideal para várias áreas de aplicação e diferentes mercados:

- Aeroespacial e Militar - Como as FPGAs são tolerantes a radiações, são uma solução para esse tipo de ambientes, com IPs (*intellectual property*) próprios para processamento digital de imagem, geração de ondas, criptografia, e reconfiguráveis parcialmente para *Software Define Radio*, etc.

- Broadcast - Soluções que permitem uma variada gama de transmissão de vídeo e áudio.

- Consumidor - Soluções rentáveis para aplicações do consumidor, dipositivos electrónicos como TVs digitais, projectores, aparelhos digitais de TV, eletrodomésticos de informação, gestão de redes de casa, etc.

- Industrial / científico / médico - Soluções para a indústria, como automatização industrial, controlo de movimento, ethernet industrial, aplicações médicas como por exemplo, aparelhos de diagnóstico, etc.

- Automóvel - Soluções de IP para sistemas de ajuda de motorista, GPS, conforto, sistemas de comunicação, processamento de gráficos e áudio. Exemplos de sistemas electrónicos num automóvel que podem ser implementados baseados em FPGA podem ser vistos na figura 3.9.

- Armazenamento/Servidores - Soluções para armazenamento de dados, servidores e computadores de grande desempenho.

- Comunicações sem fios - Aplicações em gestão de redes, em equipamento sem fios, para protocolos como WCDMA, HSDPA, WiMAX, infra-estrutura 3G, etc.

- Comunicações com fios - Soluções também para a gestão de redes, neste caso com fios, para processamento de pacotes fim-a-fim, roteamento, MAC, controladores de memória, *switch*, etc.

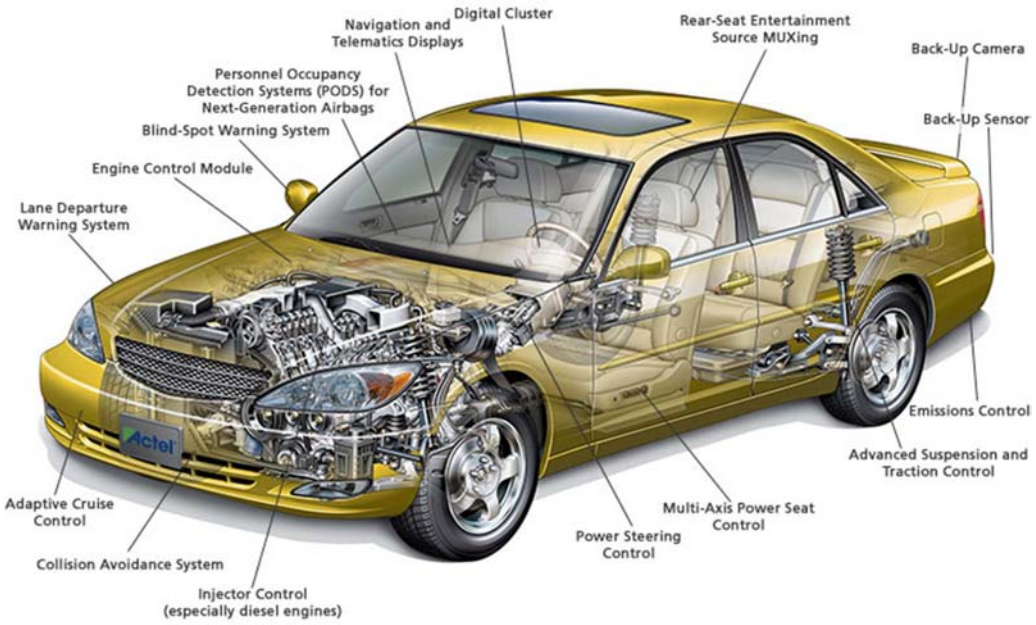


Figura 3.9: Algumas aplicações possíveis de uma FPGA num automóvel [15]..

3.2.7 Evolução

Devido a enorme evolução destas nos últimos anos, as FPGAs de hoje são cada vez mais escolhidas para todo tipo de aplicações. O maior fabricante de FPGAs actualmente, é naturalmente a Xilinx, uma vez que foram os primeiros a comercializa-la a meados da década de 80, é assim líder no mercado desde então. Esta empresa sozinha representa mais de 50% do mercado. O segundo maior fabricante é a Altera, que junto com a Xilinx controlam mais de 80% do mercado. No entanto, outros fabricantes também tem vindo a ganhar o seu espaço, como por exemplo a Lattice Semiconductor, Actel, Atmel, SiliconBlue Technologies, QuickLogic e Achronix. Esta última disponibiliza as FPGAs mais rápidas do mundo [40], pois permite velocidades de sistema até 1.5GHz, ideal para as aplicações mais exigentes em relação á velocidade.

Dentro das FPGAs disponibilizadas pela Xilinx, destacam-se duas famílias importantes: as Virtex e as Spartan. Ambas foram introduzidas no mercado em 1998 sendo estas direccionadas para situações diferentes. As Virtex, com maior capacidade e mais macro blocos heterogéneos e específicos representam a ultima geração de FPGAs, no que diz respeito ao desempenho, capacidade e integração, sendo naturalmente mais caras que as Spartan. As Spartan por sua vez, são de baixo custo, ideal assim para aplicações de grandes volumes, sendo um caminho seguro, eficiente e barato para exigências de tempo para chegar rapidamente ao mercado. A diferença das características destas duas famílias pode ser vista com mais detalhe na tabela 3.1. As FPGAs uti-

lizadas neste trabalho são da família Spartan 3 da Xilinx, devido ao seu baixo custo, e pela sua arquitectura adequada.

Features	Virtex-5	Virtex-4	Extended Spartan-3A
Logic Cells	Up to 330,000	Up to 200,000	Up to 53,000
User I/Os	Up to 1200	Up to 960	Up to 519 I/O
I/O Standards Supported	Over 40	Over 20	Over 20
Clock Management - DCM	Yes	Yes	Yes
Clock Management - PLL	Yes	No	No
Embedded Block RAM	Up to 18 Mbits	Up to 11Mbits	Up to 1.8 Mbits
Embedded Multipliers for DSP	Yes (25 x 18 MAC)	Yes (18 x 18 MAC)	Yes (18 x 18 MAC)
Multi-Gigabit High Speed Serial	Yes	Yes	No
Soft Processor Support	Yes	Yes	Yes
Embedded PowerPC Processors	Yes (PowerPC 440)	Yes (PowerPC 405)	No

Tabela 3.1: Comparação das características das FPGAs da Xilinx [23].

Um exemplo da evolução verificada nesta área, são as características nas últimas versões destas famílias de FPGAs (Virtex 6 e Spartan 6) [41]:

- Custo de sistema total reduzido até 50%;
- Redução do consumo total até 65%;
- Aumento da capacidade lógica até 760K blocos lógicos;
- Capacidade de memória em *Block RAMs* até 38 Mbit e 2000 blocos DSP;
- *Transceivers* com taxas de transmissão até 11.2 Gbps taxas consecutivas;
- Controlador de memória embutido flexível para interfaces de memória de alta velocidade;

Deste modo, com esta nova geração de FPGAs, torna-se ainda mais fácil implementar sistemas cada vez mais complexos com maior desempenho, velocidade e integração num único circuito integrado. Com estas também é possível uma redução da complexidade do PCB, consumo de energia e custo total do sistema. Com isto, devido ao aumento da complexidade dos sistemas implementados, as linguagens de descrição de hardware e os modelos IP ganham assim mais importância tendo em vista um desenvolvimento rápido do projecto.

3.3 Linguagens de descrição de hardware

As linguagens de descrição de hardware (HDLs) são orientadas à descrição estrutural e comportamental do *hardware*. Foram originalmente desenvolvidas, para documentar o comportamento dos ASICs, para substituir os complexos manuais que descreviam o funcionamento destes. Deste modo, tornaram-se uma alternativa mais válida aos diagramas esquemáticos no desenvolvimento dos circuitos digitais, com uma grande vantagem em relação a estes, a portabilidade e flexibilidade. Estas permitem projetos independentes da tecnologia e das ferramentas utilizadas, facilidade de actualização de projetos, bem como um nível mais alto de abstração. Deste modo, permite uma redução do tempo de desenvolvimento de um projeto. As HDLs juntamente com as FPGAs vieram assim revolucionar o desenvolvimento de sistemas digitais, pois permite aos *designers* desenvolver sistemas digitais bastante complexos num curto espaço de tempo. Actualmente, as HDLs mais utilizadas são o VHDL e o Verilog. A linguagem usada neste trabalho foi VHDL (de VHSIC - *Very High Speed Integrated Circuit*).

3.4 Núcleos de Propriedade Intelectual

Núcleo de Propriedade Intelectual (IP *cores*) é um bloco lógico reutilizável de lógica, que podem ser abertos ou de fabricantes específicos que oferecem versões de sintetizáveis dos blocos IP. Permitem, assim desenvolver projectos mais rapidamente, com o uso de blocos já construídos e optimizados. A Xilinx á semelhança com o que já acontece com outros fabricantes, disponibiliza vários blocos IP para utilização em sistemas com diferentes funções como:

- Áudio, vídeo e processo de imagem;
- Automóvel;
- Lógica Básica;
- Interface de bus e IO;
- Comunicação em rede;
- Processo digital de sinal;
- Matemática;
- Interface de memória;

- Armazenamento de dados;
- Processadores;

Por exemplo, neste trabalho foram utilizados o *Fifo Generator v4.4* e o *Block Memory Generator v2.8* da Xilinx LogiCORE™ IP para a implementação de *fifos* e de blocos de memória, respectivamente.

3.5 Ferramentas de suporte ao desenvolvimento

Existem vários programas que suportam este tipo de desenvolvimento, no entanto o fluxo de projecto é essencialmente o mesmo. A ferramenta utilizada neste trabalho foi o ambiente de síntese integrado da Xilinx, o ISE WebPACK, devido a ser uma ferramenta livre, e por ter todas funcionalidades necessárias a realização deste trabalho, não havendo necessidade de utilização de outros programas mais caros com maior desempenho (como por exemplo Synplicity), uma vez que o objectivo principal deste projecto é verificação de funcionalidades e simulação mas não optimização.

Uma ferramenta de síntese de boa qualidade não só entrega melhor desempenho e confiança quando comparada a outras ferramentas de síntese, mas também uma grande ajuda em termos de verificação e *debug*, o que é extremamente importante para implementação mais rápida de sistemas inevitavelmente cada vez mais complexos. Por isso, estas ferramentas são normalmente bem mais caras. Um exemplo destas ferramentas, é a empresa Synplicity Inc, que é um fornecedor de soluções de *software* para design e implementação de dispositivos de lógica programáveis (FPGAs, PLDs e CPLDs) para as aplicações mais exigentes em que as suas ferramentas apresentam melhores desempenhos e velocidades que as ferramentas disponibilizadas pelos fornecedores de FPGAs.

3.5.1 Xilinx Integrated Software Environment (ISE)

O ISE WebPACK é assim uma solução para o projecto de um sistema em FPGA ou CPLD da Xilinx, pois fornece todas as ferramentas necessárias para a síntese de HDL, simulação, implementação e programação do dispositivo respectivo, neste caso uma FPGA. Na figura seguinte é possível observar a sequência de etapas necessárias no fluxo de um projecto desenvolvido no ISE.

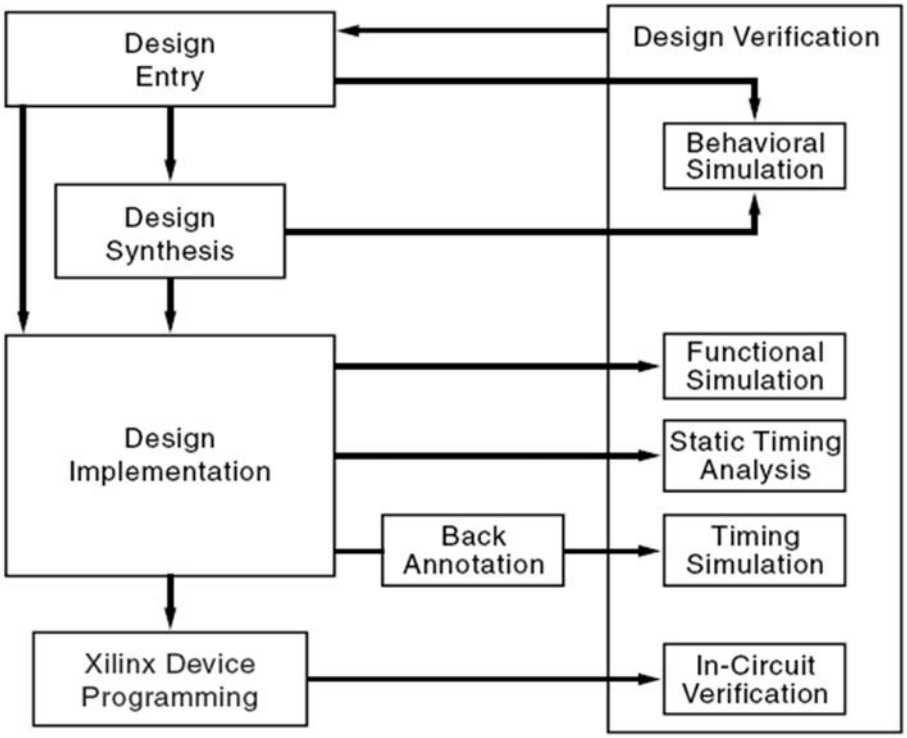


Figura 3.10: Fluxo de desenvolvimento de um projecto no Xilinx ISE[16].

Assim, as etapas principais no desenvolvimento de um projecto são as seguintes:

- **Desenho do sistema (*Design Entry*)** - Descrição do sistema usando uma linguagem HDL, aqui serão adicionados todos os ficheiros de código necessários á especificação do sistema. Podem também ser adicionados blocos já previamente construídos, bem como blocos das livrarias IP(*Intellectual Property*) existentes caso sejam utilizados. Por fim, é necessária a inclusão do ficheiro UCF (*User Contrain File*) que especifica os constrangimentos da implementação, nomeadamente os pinos de entrada e saída correspondentes na FPGA.

- **Síntese (*Design Synthesis*)** - Depois de feita a análise da sintaxe do código anterior, se não houver erros, o sistema é sintetizado. Este processo é onde o software transforma o código HDL em componentes lógicos, como por exemplo simples portas lógicas e *flip-flops*. Embora muitos códigos HDL sejam correctos em termos de sintaxe, não significa que sejam sintetizáveis, pois não são possíveis de implementar fisicamente. Estes códigos só podem assim ser utilizados para simulação comportamental.

- **Implementação (*Design Implementation*)** - Este processo consiste em três sub-processos: *translate*, *map* e *place and route*. No processo *translate*, é com-

binada toda a informação fornecida pela síntese num único ficheiro netlist. No processo *map*, é feito o mapeamento dos componentes lógicos contidos no netlist nos recursos da FPGA, respectivamente nos blocos lógicos e nos blocos de IO descritos anteriormente. Por fim, no processo *place and route*, é feita a colocação da lógica nos locais físicos respectivos, e a determinação das rotas para a interligação entre os vários sinais do sistema. No final da implementação, através da análise temporal, são determinados vários parâmetros importantes como o atraso máximo de propagação que existe nos circuitos implementados, que vai influenciar directamente a frequência máxima de relógio, para o correcto funcionamento do sistema implementado.

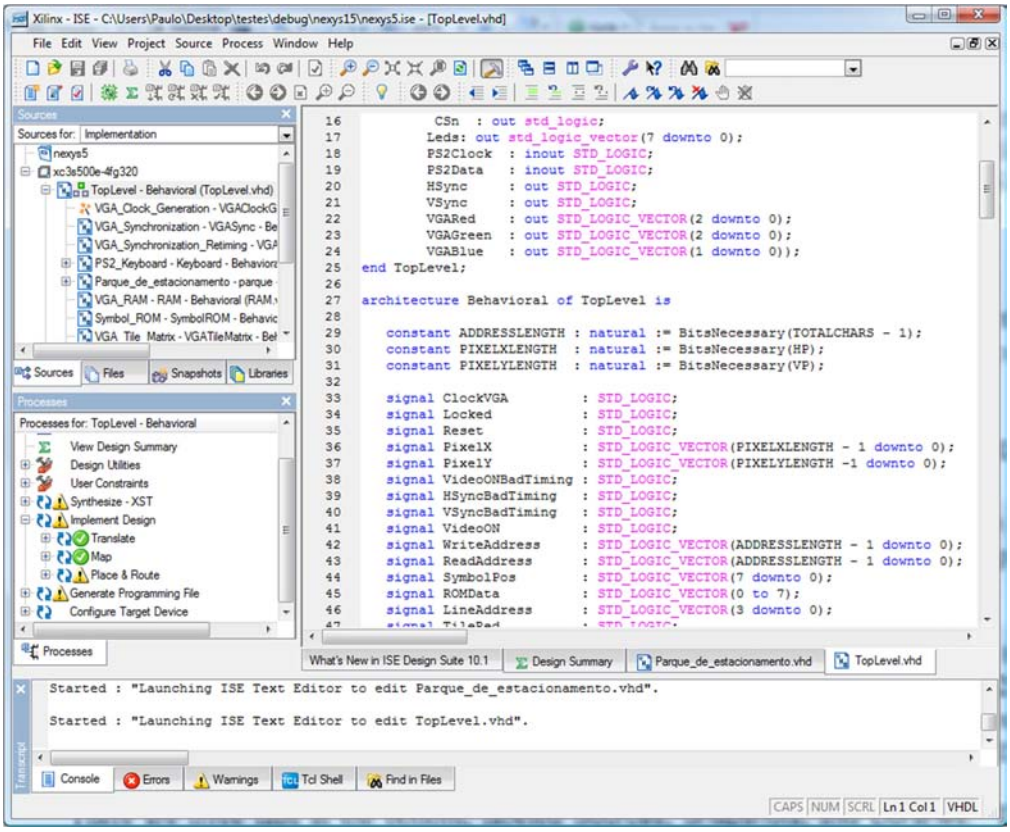
- **Programação da FPGA (*Xilinx Device Programming*)** - Aqui é gerado o ficheiro de configuração final (*.bit) e em seguida carregado em série para a FPGA, através de um cabo JTAG próprio ou por uma simples ligação USB. A maioria das placas de desenvolvimento com FPGAs contêm uma PROM (não volátil) que permite guardar este *bitstream*, programando assim as FPGAs automaticamente sempre que as placas são ligadas.

Como se pode verificar através da figura 3.10, todas as etapas descritas anteriormente são acompanhadas com simulação e verificação do sistema implementado, a nível comportamental, funcional e temporal, facilitando assim o desenvolvimento do sistema e a detecção de erros, bem como a resolução dos mesmos.

O Xilinx ISE controla todos os aspectos do fluxo de desenvolvimento oferecendo uma interface gráfica para todas a ferramentas disponíveis e ficheiros associados com o projecto. Como se pode observar na figura 3.11, o ambiente de trabalho por defeito é dividido em quatro secções:

- *Sources window* (em cima á esquerda): mostra hierarquicamente todos os ficheiros associados ao projecto. Aqui também é possível especificar o objectivo, se é simulação ou síntese e implementação, no topo desta janela ("*Sources for:*"). Esta secção também oferece a possibilidade de guardar cópias de segurança do projecto, bem como a verificação das bibliotecas associadas a este.

- *Processes window* (no meio á esquerda): Nesta secção são mostrados todos os processos disponíveis ao ficheiro seleccionado na secção anterior. Alguns processos podem conter sub-processos, como no caso da implementação. O ISE executa alguns processos automaticamente, necessários a um determinado processo, como por exemplo para a implementação, ele executará primeiro automaticamente a síntese, uma vez que a implementação e depende do resultado desta. Deste modo,




```
16          CSn  : out std_logic;
17         Leds: out std_logic_vector(7 downto 0);
18         PS2Clock  : inout STD_LOGIC;
19         PS2Data   : inout STD_LOGIC;
20         HSync     : out STD_LOGIC;
21         VSync     : out STD_LOGIC;
22         VGARed    : out STD_LOGIC_VECTOR(2 downto 0);
23         VGAGreen  : out STD_LOGIC_VECTOR(2 downto 0);
24         VGABlue   : out STD_LOGIC_VECTOR(1 downto 0));
25 end TopLevel;
26
27 architecture Behavioral of TopLevel is
28
29     constant ADDRESSLENGTH : natural := BitsNecessary(TOTALCHARS - 1);
30     constant PIXELXLENGTH  : natural := BitsNecessary(HP);
31     constant PIXELYLENGTH  : natural := BitsNecessary(VP);
32
33     signal ClockVGA         : STD_LOGIC;
34     signal Locked           : STD_LOGIC;
35     signal Reset            : STD_LOGIC;
36     signal PixelX           : STD_LOGIC_VECTOR(PIXELXLENGTH - 1 downto 0);
37     signal PixelY           : STD_LOGIC_VECTOR(PIXELYLENGTH -1 downto 0);
38     signal VideoONBadTiming : STD_LOGIC;
39     signal HSyncBadTiming   : STD_LOGIC;
40     signal VSyncBadTiming   : STD_LOGIC;
41     signal VideoON          : STD_LOGIC;
42     signal WriteAddress     : STD_LOGIC_VECTOR(ADDRESSLENGTH - 1 downto 0);
43     signal ReadAddress      : STD_LOGIC_VECTOR(ADDRESSLENGTH - 1 downto 0);
44     signal SymbolPos        : STD_LOGIC_VECTOR(7 downto 0);
45     signal ROMData          : STD_LOGIC_VECTOR(0 to 7);
46     signal LineAddress      : STD_LOGIC_VECTOR(3 downto 0);
```

Figura 3.11: Ambiente típico do software integrado ISE da Xilinx.

o ISE garante a passagem por todas as etapas necessárias ao desenvolvimento de um projecto, descritas anteriormente.

- *Transcript window* (em baixo): Aqui é mostrada toda a informação do progresso do processo em execução, avisos e erros. Através da aba "*Tcl Shell*" também é possível executar processos através de comandos específicos numa linha de comandos.

- *Workplace window* (em cima á direita): Esta é a janela principal (normalmente maior), que permite conter vários ficheiros abertos e a edição destes, em código HDL ou esquemáticos, ou a visualização de documentos importantes, como por exemplo relatórios da síntese e implementação do projecto. Através destes documentos é possível identificar alguns problemas e soluções, também é possível verificar a percentagem usada dos recursos da FPGA, bem como os recursos usados por cada módulo individualmente.

Todas estas janelas podem ser redimensionadas e movidas conforme o gosto do utilizador. Esta ferramenta está disponível tanto para Windows como para Linux. O

Xilinx ISE suporta também integração com outros programas como o MATLAB, o Synplify e o ModelSim. Apesar do ISE já conter um simulador embutido, é possível através deste chamar um simulador externo mais versátil e robusto, o ModelSim fabricado pela Mentor Graphics Corporation, sendo assim uma ferramenta de *software* independente do ISE que permite ao utilizador simular um determinado sistema digital antes de o implementar em hardware.

3.5.2 ModelSim

Á medida que aumenta a complexidade dos sistemas digitais, a sua verificação e debug torna-se cada vez mais difícil. Neste contexto, simuladores como o ModelSim ganham maior importância, uma vez que permite uma análise e verificação mais detalhada do comportamento de todo o sistema, através de diagramas temporais dos sinais do sistema, facilitando assim resolução de problemas. Deste modo é obtido um melhor desempenho do sistema e um desenvolvimento mais rápido deste.

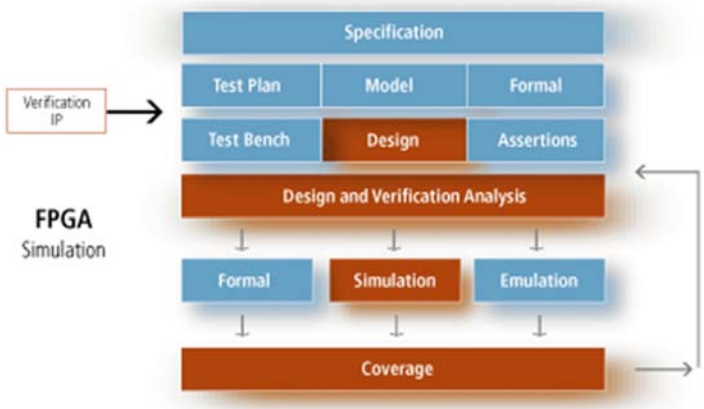


Figura 3.12: Fluxo de desenvolvimento de um sistema no modelsim[17].

O ModelSim fornece assim um ambiente indicado de simulação e *debug* para projectos de complexidade elevada em ASIC e em FPGA, além disso também inclui suporte para várias linguagens como VHDL, Verilog, SystemVerilog e SystemC [17]. Este *software* está também disponível tanto para Windows como para Linux. Este simulador está assim, em diferentes edições, integrado nos vários *softwares* dos maiores fabricantes de FPGA, entre eles, a Xilinx, altera, atmel, actel.

Estas foram as ferramentas utilizadas neste trabalho (ISE e Modelsim), pelos motivos já referidos anteriormente. Ambas tem uma documentação basta e detalhada que pode ser consultada no menu de ajuda ou nos seus sites [42, 16] para mais informação.

3.6 Placas de desenvolvimento

As placas de desenvolvimento utilizadas neste trabalho foram a Nexys 2 da Digilent e RC10 da Celoxica, devido ao seu preço mais baixo, e porque contém todos os requisitos necessários para a execução deste trabalho. No entanto existem placas com muitos mais recursos de vários fornecedores, sendo naturalmente bem mais caras.

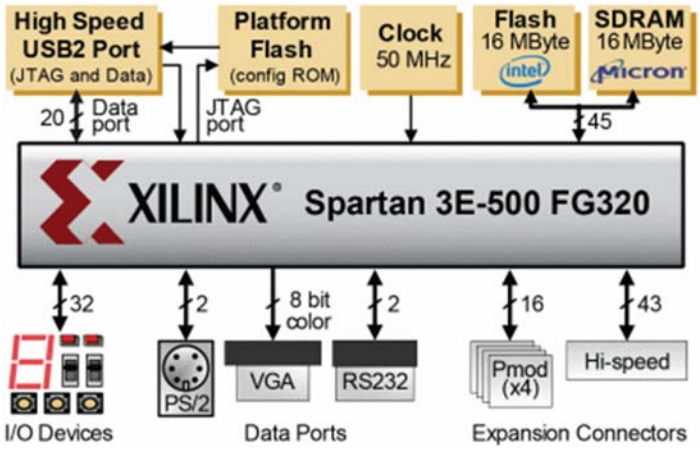


Figura 3.13: Diagrama de blocos da placa Nexys 2[18].

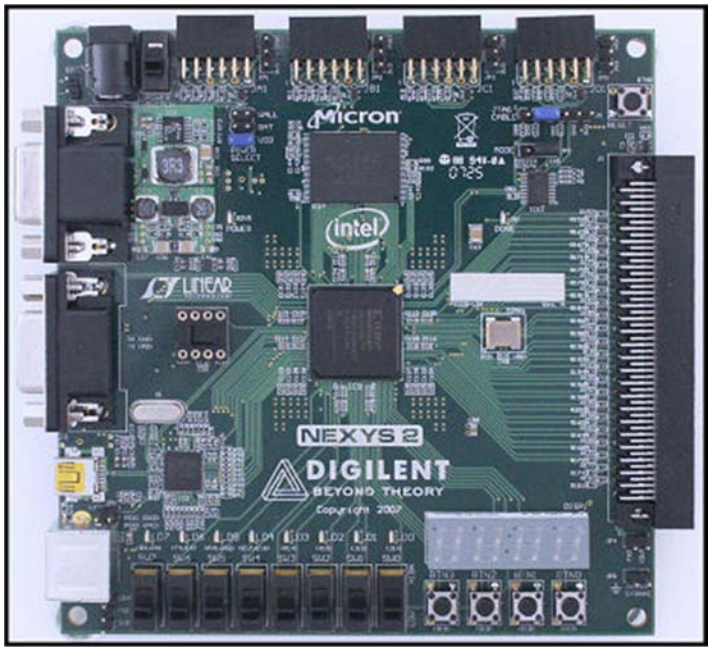


Figura 3.14: Nexys 2[18].

A placa Nexys2 tem as seguintes características [18]:

- FPGA Xilinx Spartan 3E com 500 mil portas lógicas;

- Configuração através de uma ligação USB2 de alta-velocidade;
- Alimentação por USB, bateria ou simples cabo;
- Memórias externas de capacidade 16MB SDRAM da Micron e 16MB *Flash* da Intel;
- Uma PROM *Flash* não volátil para configurar a FPGA;
- Oscilador externo 50MHz;
- Interfaces PS/2, VGA (8 bits de cor) e RS232;
- 60 ligações I/O da FPGA para conectores de expansão;
- 8 LEDs, 4 *displays* de 7 segmentos, 4 botões, 8 interruptores.

A placa RC10 da Celoxica apresenta mais recursos que a placa anterior, em termos de interfaces como na capacidade lógica da FPGA embutida nela. Esta contém uma FPGA da Xilinx Spartan-3 com 1,5 milhões de portas lógicas, ou seja, o triplo da anterior.

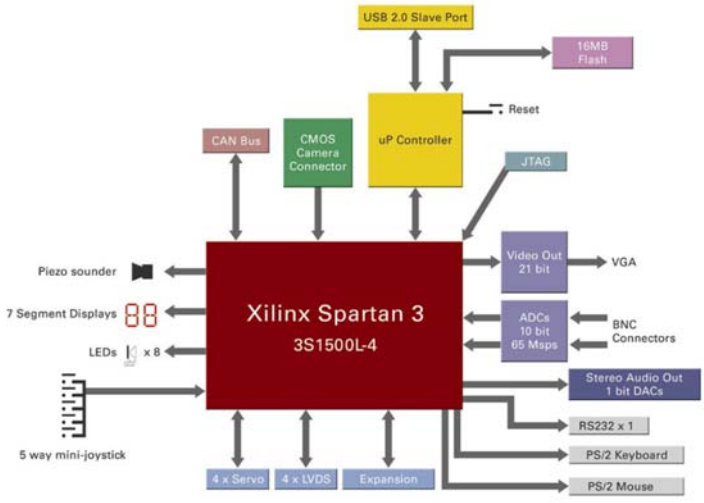


Figura 3.15: Diagrama de blocos da placa RC10 da celoxica[19].

A figura 3.15 ilustra o diagrama de blocos desta placa com os seus componentes e interfaces principais.

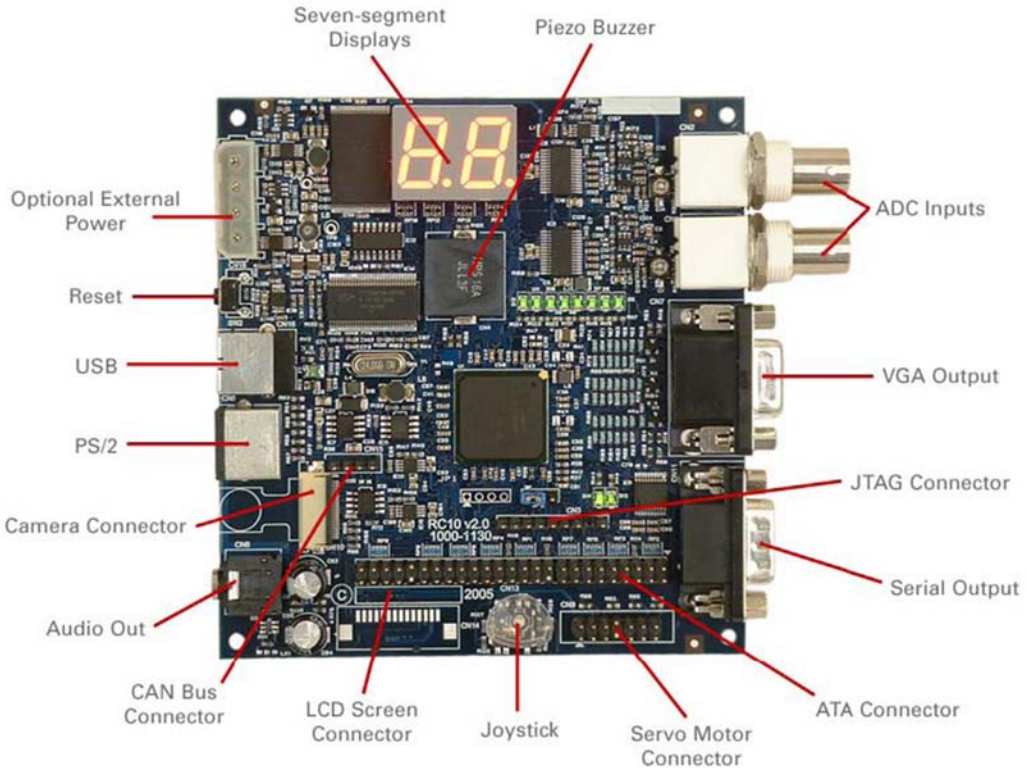


Figura 3.16: Celoxica RC10[19].

3.7 Interacção remota para sistemas de controlo

O principal objectivo deste trabalho é introduzir uma interface que permita a uma FPGA comunicar com o mundo externo remotamente, isto é, estabelecer uma ligação sem fios com outras FPGAS, ou outros tipos de sistemas computacionais. Para isso utilizou-se um dos vários *transceivers* RF que existem disponíveis actualmente no mercado de modo a satisfazer os objectivos que se pretende para esta ligação sem fios:

- Bidireccional;
- Confiável;
- Baixo custo;
- Baixo consumo de energia;
- Baixa interferência;
- Distância razoável (na ordem das dezenas de metros);
- Velocidade de transferência de dados na ordem de vários kbits por segundo;

Os *transceivers* RF normalmente são controlados por um microprocessador, pois alguns dos fornecedores destes *transceivers* RF já disponibilizam também o código próprio em *software* para a programação das ligações sem fios.

Neste caso, este será ligado directamente á FPGA, isto é, a comunicação com o *transceiver* e a sua programação será feita toda através dos recursos disponíveis em qualquer FPGA actual, ou seja, através da programação de hardware com o uso da linguagem VHDL.

Com isto, pretende-se desenvolver um módulo VHDL que possa ser reutilizável numa basta gama de projectos em FPGAs que necessitem de comunicação sem fios, sem a necessidade de o uso de um microprocessador para isso e assim o controlo seja efectuado todo através de hardware.

3.7.1 *Transceiver* RF

Actualmente, existem disponíveis no mercado vários *transceivers* RF que podem ser usados para estabelecer uma ligação com as características anteriormente descritas. No entanto, após uma breve pesquisa sobre várias marcas de fornecedores observou-se que o *transceiver* CC1101 da Texas Instruments é o mais apropriado para esta aplicação, pois é o que oferece uma melhor relação qualidade preço, sendo superior na maioria das características em relação aos outros *transceivers* com um preço idêntico a este, como se pode ver em comparação por exemplo em [43, 44, 45].

3.7.2 Características do *transceiver* CC1101

Deste modo, as características que determinaram a escolha deste *transceiver* para este projecto são as seguintes:

- Elevada sensibilidade (-112dBm a 1.2 kBaud, 1% de taxa de erro);
- Baixo consumo (14.7 mA na recepção);
- Potência de saída programável até 10 dBm;
- Taxa de transferência programável (desde 1.2 a 500 kBaud);
- Basta gama de frequências: 300-348 MHz, 387-464 MHz e 779-928 MHz;
- Vários tipos de modulações digitais: 2-FSK, GFSK, MSK, OOK, ASK;
- Ideal para sistemas com *Frequency Hopping*;

- Compensação automática da frequência para um alinhamento correcto com a portadora recebida;
- Suporte para pacotes de dados com tamanho variável;
- Detecção automática de endereços, palavras de sincronização, correcção de erros e CRC;
- Indicador CCA (*Clear Channel Assessment*);
- Indicador RSSI;
- Interface com o módulo através do protocolo SPI;
- Fifos de recepção de transmissão de 64 bytes cada;
- Wake-On-Radio (WOR).

3.7.3 Comunicação com o módulo de RF - Protocolo SPI

Deste modo, para a sua programação e ligação do *transceiver* RF escolhido com o resto do sistema, ou seja, com a FPGA é utilizada uma interface SPI (*Serial Peripheral Interface*) para esse efeito. O protocolo SPI foi desenvolvido pela Motorola, com o objectivo de fornecer uma interface simples entre os microcontroladores e periféricos externos a este. Este protocolo é síncrono, pois todas as transmissões são controladas através de um relógio comum, gerado pelo *master* que geralmente na maioria das aplicações costuma ser um processador. Podem ser ligados vários *slaves* através da interface SPI ao mesmo *master*, no entanto só um pode estar activo de cada vez, seleccionado pelo *master* através do sinal CS (*chip select*) respectivo.

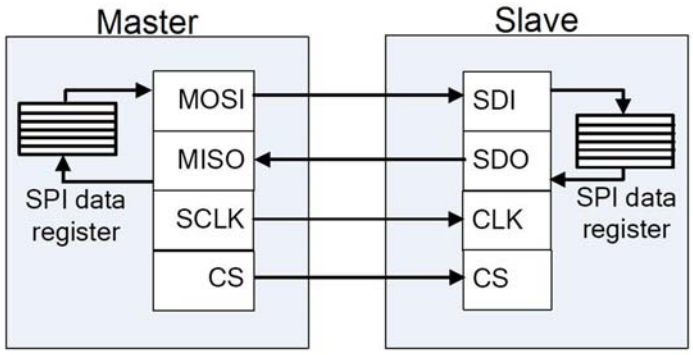


Figura 3.17: Comunicação SPI entre dois dispositivos (adaptado[4]).

Esta comunicação é *full-duplex*, ou seja, existe comunicação nos dois sentidos. Para isto, o protocolo SPI utiliza 4 sinais para o controlo e transferência de dados:

- *Master Output Slave Input* (MOSI): Sinal gerado pelo *master* que envia os dados para o *slave*, também conhecido por *Slave Data In* (SDI);

- *Master Input Slave Output* (MISO): Sinal gerado pelo *slave* que envia os dados para o *master*, também conhecido por *Slave Data Out* (SDO);

- *Serial Clock* (SCLK ou SCK): Sinal de relógio gerado pelo *master* que controla e sincroniza a transferência de dados;

- *Chip Select* (CS): Sinal gerado pelo *master* que activa e selecciona o *slave*, também conhecido *Slave Select*(SS).

Este protocolo de comunicação em série, torna-se bastante eficiente, pois á medida que o *slave* lê dados (bits) na linha MOSI, ele simultaneamente escreve dados também pela linha MISO. Deste modo ambos recebem e enviam informação ao mesmo tempo.

Este protocolo apresenta quatro modos de operação, dependendo da polaridade e fase do relógio, podendo cada um estar em dois estados diferentes. Estes modos determinam em que altura a informação escrita e lida nas linhas respectivas (MISO e MOSI). Um bit pode assim ser escrito na fase ascendente do relógio (quando o relógio passa de 0 para 1) e lido na fase descendente do relógio (quando o relógio passa de 1 para 0) ou vice-versa, dependendo do modo de operação. Estes modos de operação são incompatíveis um com o outro, por isso o *master* e o *slave* devem estar configurados no mesmo modo para poderem comunicar um com o outro.

Assim, devido a flexibilidade e eficiência deste protocolo, ele é utilizado numa basta gama de aplicações. Este pode ser utilizado para fazer a interface com sensores, memórias, ADCs, DACs, LCDs, dipositivos de controlo, circuitos de áudio, em comunicações, com outros processadores, em FPGAs, etc.

3.7.4 Modo de funcionamento do *transceiver* CC1101

Este módulo combina um transmissor e um receptor no mesmo circuito integrado, característica fundamental para uma comunicação nos dois sentidos, permitindo deste modo utilizar componentes comuns necessários a transmissão e recepção.

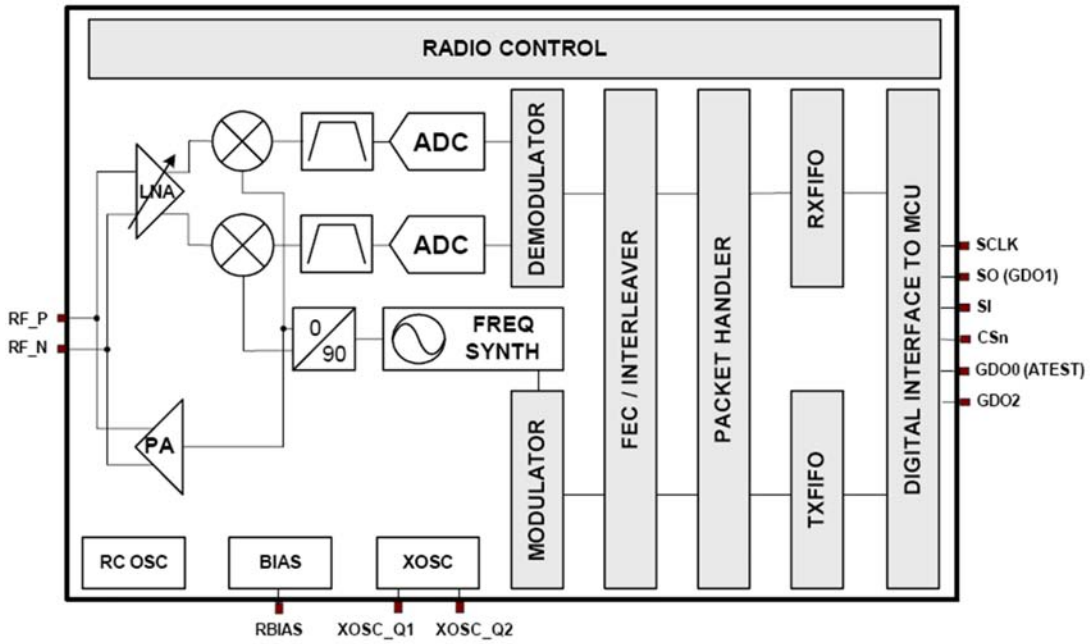


Figura 3.18: Diagrama de blocos simplificado do transceiver RF utilizado (cc1101) [20].

Na recepção o sinal RF recebido passa pelo amplificador de baixo ruído LNA (*low-noise amplifier*) onde é amplificado para níveis aceitáveis de forma a reduzir o seu ruído para uma melhor recuperação do sinal para o resto do circuito de recepção. Depois disto, o sinal RF é convertido numa frequência intermédia (IF) mais baixa. Os sinais em fase e quadratura (I/Q) resultantes são digitalizados pelas ADCs, e a partir daqui o restante processamento como desmodulação, correctores de erro e processamento de pacote é feito digitalmente. Por fim a informação recebida é colocada na fifo de recepção onde pode ser acedida pelo *master* do protocolo SPI, que neste caso será uma FPGA.

Na transmissão, os dados que precisam ser enviados remotamente são escritos na *fifo* de transmissão através do protocolo SPI. De seguida, os dados serão processados digitalmente através de vários blocos como o gestor de pacotes, o gestor de erros, e depois pelo modulador digital. São geradas as componentes I/Q e por fim, o sinal gerado é amplificado pelo amplificador PA.

3.7.5 Configuração

Das frequências livres que podem ser utilizadas, a frequência 2.4 GHz já é bastante utilizada por outras tecnologias como o *WiFi* e o *Bluetooth*. Por isso para este caso as frequências livres abaixo de 1 GHz são mais adequadas, pois é menor a probabilidade de interferência com outras tecnologias. Deste modo, a frequência utilizada neste trabalho

será no intervalo 387-464MHz uma vez que neste intervalo de frequências a ocupação do espectro é relativamente mais baixa e permite assim uma maior distância da ligação sem fios em relação às outras frequências possíveis para este transceiver, uma vez que quanto menor a frequência, maior pode ser a distância de comunicação sem fios. Para o funcionamento do *transceiver* CC1101, este terá de ser colocado num circuito como o que pode ser visto na figura 3.19, de modo a haver uma correcta adaptação com a antena que realmente vai fazer a interface com o meio sem fios.

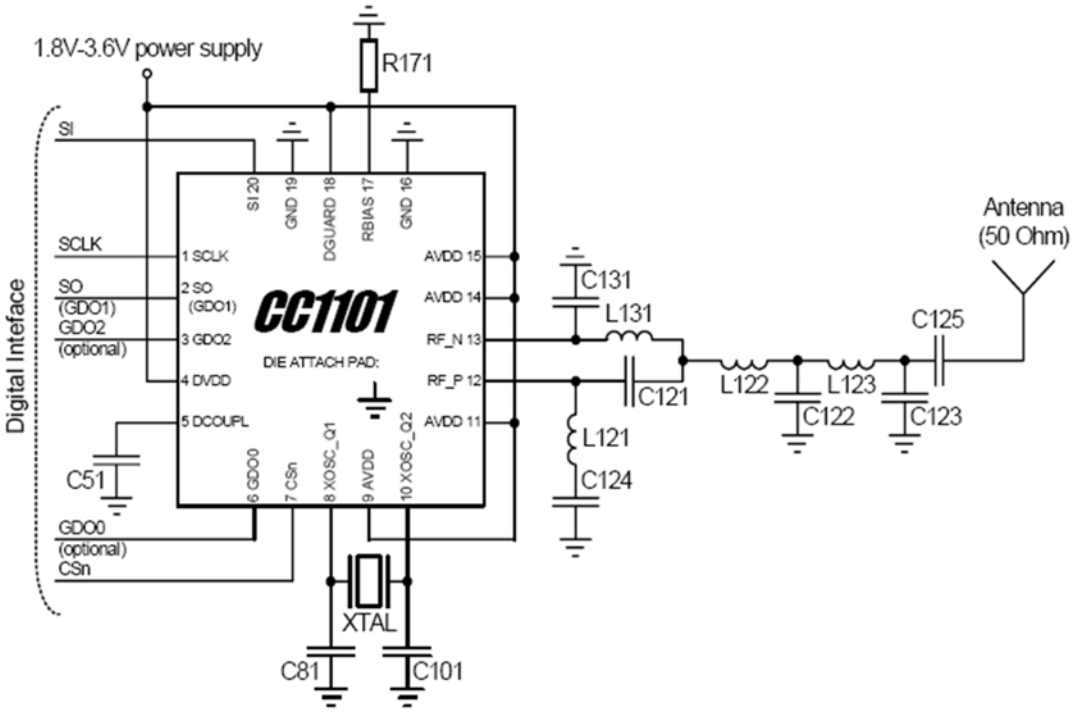


Figura 3.19: Circuito típico para o funcionamento do transceiver nas frequências 315/433 MHz[20].

Para isto, devido as pequenas dimensões do *transceiver* CC1101 (4x4 mm), neste trabalho utilizou-se um módulo (figura 3.20) que já possui este *transceiver* colocado num circuito impresso com toda a malha de adaptação necessária.

Uma das grandes vantagens deste transceiver, é o facto deste poder ser bastante configurável através da interface SPI, pois pode ser adaptado a diferentes tipos de ligações sem fios, através da alteração dos seus registos. Podendo assim ter um bom desempenho para muitos sistemas com diferentes aplicações, um pouco a semelhança com o que acontece com as FPGAs. Deste modo, este *transceiver* é uma boa solução para sistemas que utilizam FPGAs. Os parâmetros principais que podem ser alterados neste *transceiver* são:

Figura 3.20: *Evaluation module* CC1101.

- Taxa de transmissão;
- Tipo de modulação;
- Potência de saída;
- Canal de Recepção;
- Largura de banda do filtro de recepção;
- Modo de transmissão/recepção;
- Formato dos pacotes de dados;
- Gestão das *fifos* de recepção e de transmissão;
- Influência do CCA no modo de transmissão;
- Correcção de erros (FEC);
- Wake-On-Radio (WOR).

Para a correcta configuração dos registos, o fornecedor do *transceiver* cc1101 tem disponível um *software*, o SmartRF Studio[46], que permite calcular automaticamente os valores que devem ser escritos nos respectivos registos de configuração consoante o tipo de ligação que o utilizador pretende implementar. Deste modo, através de uma interface gráfica oferecida por este programa como pode ser vista na figura 3.21, é possível seleccionar todas as opções onde o *transceiver* é programável, para a obtenção dos valores óptimos para uma dada configuração.

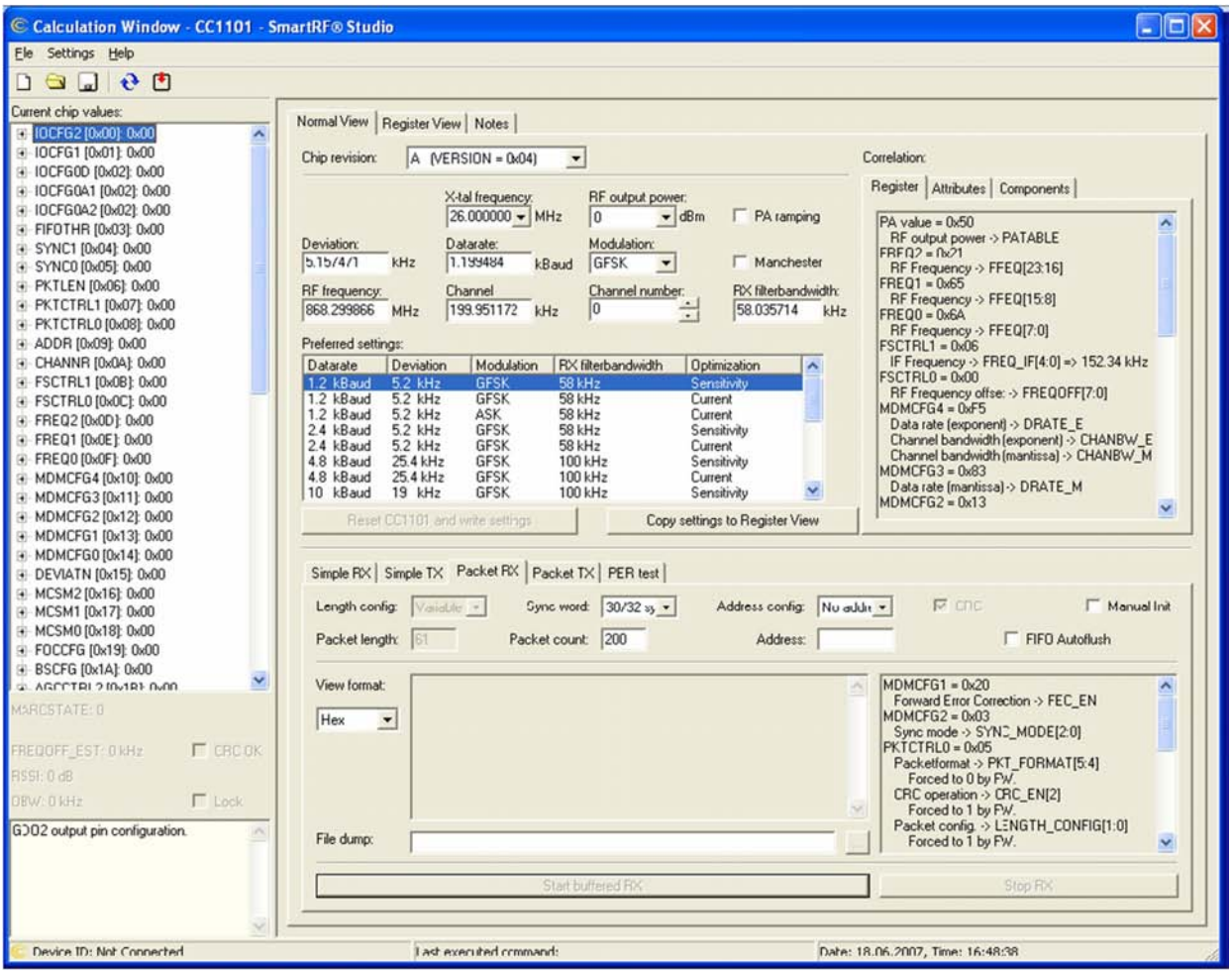


Figura 3.21: SmartRF Studio.

Capítulo 4

Sistemas implementados

Sumário

Neste capítulo é feita a descrição detalhada da implementação prática dos sistemas embutidos construídos, mais especificamente, o *buffer* de prioridade construído através de um estrutura baseada numa árvore binária, um sistema próprio para teste deste tipo de *buffer* e finalmente um sistema embutido para controlo automático de uma parque de estacionamento.

4.1 *Buffer* de prioridade

O *buffer* implementado baseado em [47], recebe à entrada informação de uma forma sequencial e fornece na saída a entrada com maior prioridade que se encontra naquele momento na memória, ordenando assim a sequência das saídas sempre que recebe uma nova entrada, colocando-a na sua posição respectiva de prioridade. O *buffer* de prioridade permite assim guardar todos os valores de chegada em memória, fornecendo ao sistema qual o próximo valor que deve ser processado por este segundo uma ordem de prioridade especifica.

4.1.1 Estrutura de dados

Uma estrutura possível para a implementação de um *buffer* de prioridade num sistema como o parque de estacionamento, no caso mais simples onde cada lugar têm uma prioridade fixa baseada na proximidade da entrada ou saída do parque, seria por exemplo criar um simples vector binário que indicasse os lugares ocupados e os lugares livres. Neste caso a procura do lugar com maior prioridade seria apenas percorrer este vector

desde o primeiro lugar até encontrar o primeiro lugar livre.

No entanto, esta solução apresenta limitações, pois impõem prioridades fixas e o número de entradas conhecido a partida, o que limita a possível utilização para sistemas mais complexos. Com isto, pretende-se desenvolver um *buffer* de prioridade reutilizável com capacidade de fornecer uma gestão mais complexa, de modo que seja possível a sua adaptação para alterações dinâmicas nas prioridades e onde o número de entradas possível não seja fixo.

Para isto, listas ligadas de prioridades e árvores binárias são estruturas mais próprias para esse efeito como se pode observar em [1]. Com estes tipos de estrutura, apesar de mais complexas, o *buffer* de prioridade apresenta um melhor desempenho e pode ser adaptado em vários tipos de sistemas mais complexos, como por exemplo em redes de comunicação ou em sistemas computacionais de alto desempenho que processam tarefas com prioridades dinamicamente variáveis.

O modelo deste *buffer* implementado na prática baseia-se numa árvore binária. Esta árvore é organizada de maneira a que para um dado nó, a sub-árvore à direita só contenha valores superiores a este, enquanto que a sub-árvore a esquerda só contém valores inferiores. No caso específico de quanto maior for o número da entrada menor será a sua prioridade, para retirar a entrada com maior prioridade que o *buffer* contém, ou seja, o menor número, será necessário percorrer toda a árvore de maneira a escolher sempre a sub-árvore a esquerda, até encontrar um nó que não contenha nenhum nó filho à esquerda, sendo que o valor deste nó será o mais pequeno do buffer, logo, a entrada com maior prioridade.

Por exemplo, considerando que no início o *buffer*, ou seja, a árvore está vazia e para uma sequência de entradas com a seguinte ordem: 23,4,6,76,82,5,99,1,3 e por fim 75, teremos a seguinte árvore binária que se pode verificar na figura 4.1.

A memória deste *buffer* é implementada em *hardware* através de um bloco de memória fixo com capacidade para um determinado número de nós. A memória fica assim organizada em blocos com a seguinte informação para cada nó:

- Valor: número que define a prioridade do nó;
- Ocupação: quando está a '1' significa que o bloco está ocupado por um nó, enquanto que quando está a '0' o bloco está livre;
- Nó pai: número que indica o índice na memória onde se encontra o nó acima deste, ou seja o seu nó pai; quando este número está a zero num dado nó significa que este é o nó raiz da árvore pois é o único nó em toda a árvore que não tem nó pai;

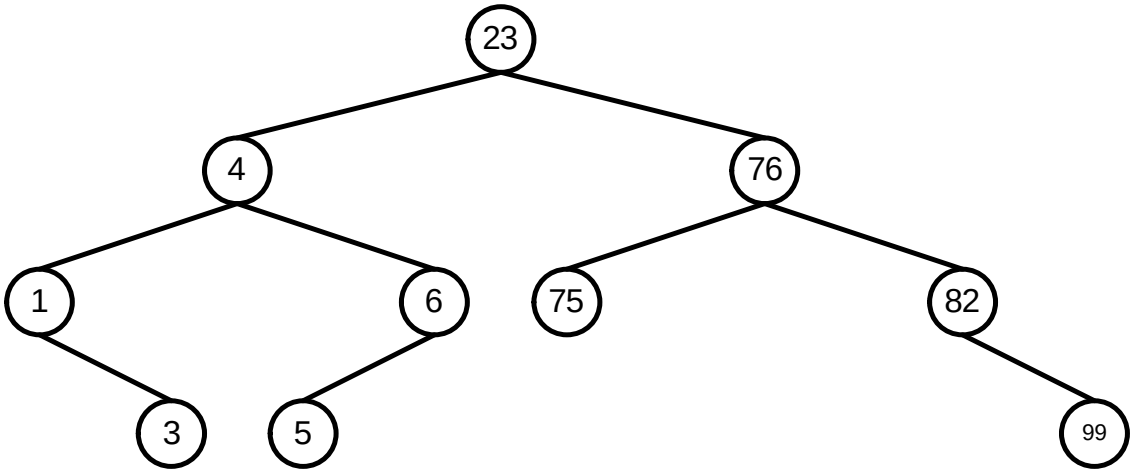


Figura 4.1: Exemplo de uma árvore binária.

- Nó filho a direita: número que indica o índice na memória onde se encontra o nó á direita deste; caso este número seja igual a zero significa que este nó não existe;
- Nó filho a esquerda: número que indica o índice na memória onde se encontra o nó á esquerda deste; caso este número seja igual a zero significa que este nó não existe;

O índice da memória começa no número um, assim quando se encontra o número zero o respectivo nó não existe. Esta memória é ocupada sequencialmente de maneira a que um qualquer novo nó que se adicione à árvore ocupe o primeiro bloco livre, ou seja, o bloco com campo ocupação igual a zero, desde o início da memória como se pode verificar mais a frente no módulo Z7.

Quando se remove um nó da árvore, o respectivo campo ocupação é igualado a zero podendo a partir daí esse bloco de memória ser utilizado outra vez por um novo nó. Deste modo, os blocos desta memória são ocupados e libertados dinamicamente em *hardware*. Assim para o exemplo da árvore anterior vamos ter o seguinte conteúdo na memória:

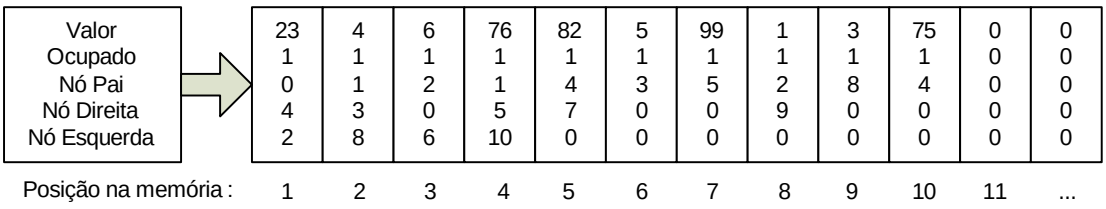


Figura 4.2: Conteúdo da memória no caso do exemplo anterior.

Como para cada nó da árvore o procedimento é o mesmo, torna-se útil o uso de

algoritmos recursivos, no entanto como as linguagens de descrição de hardware não suportam directamente chamadas recursivas, utilizou-se para isso uma máquina de estados finitos hierárquicas (HFSM), permitindo assim chamadas recursivas e hierárquicas em *hardware*, ou seja, qualquer módulo pode activar-se a si próprio. Para o controlo deste *buffer* foi utilizado assim uma máquina de estados finitos hierárquica paralela, devido a necessidade de adição e remoção de nós da árvore em paralelo, permitindo assim chamadas em paralelo de módulos diferentes, através de uma HFSM com múltiplas *stacks* (uma *stack* por cada módulo em paralelo). Deste modo, vamos ter duas *stacks* em paralelo, uma para adicionar e outra para remover nós da árvore.

O acesso a memória é devidamente sincronizado para que dois módulos das duas *stacks* possam ser executados em paralelo. Os módulos mais básicos e importantes que permitem a construção da árvore binária e a remoção dos nós da árvore são Z4 e Z5 respectivamente, os outros módulos fazem operações suplementares e sincronizam o acesso à memória partilhada. Para que as operações sejam executas desde a raiz da árvore, o índice da memória que contém este nó raiz da árvore deve ser guardado num registo para esse efeito. Assim foram utilizados 8 módulos para contrução deste buffer, cuja função é :

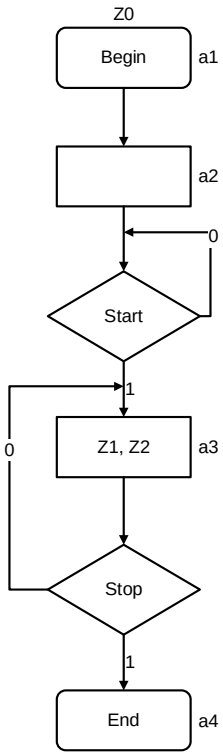


Figura 4.4: Diagramas de fluxo do módulo Z0.

- **Z0** – Implementa o algoritmo de topo ("*Top level*"), e activa dois módulos em

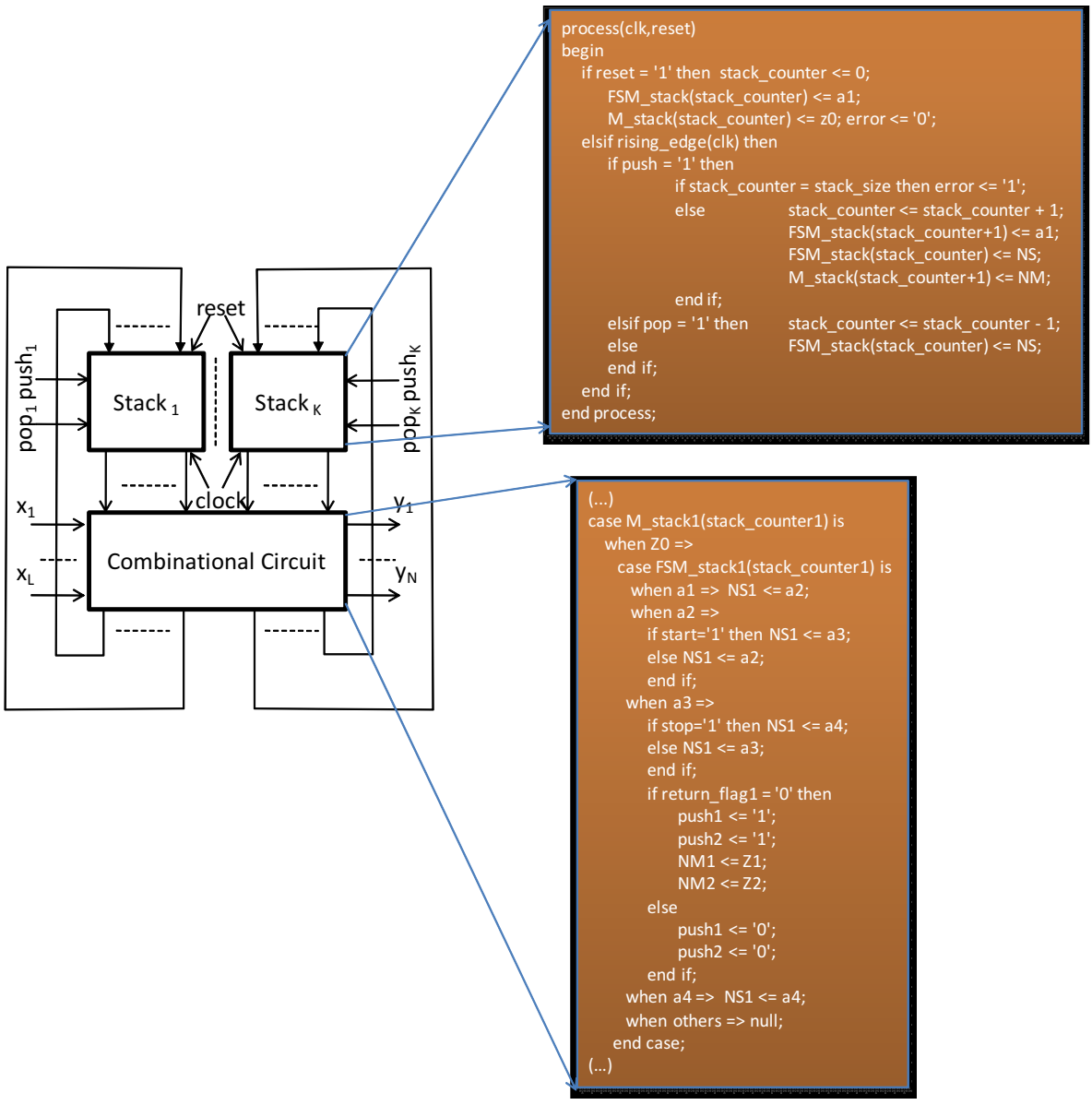


Figura 4.3: Máquina de estados finitos hierárquica paralela.

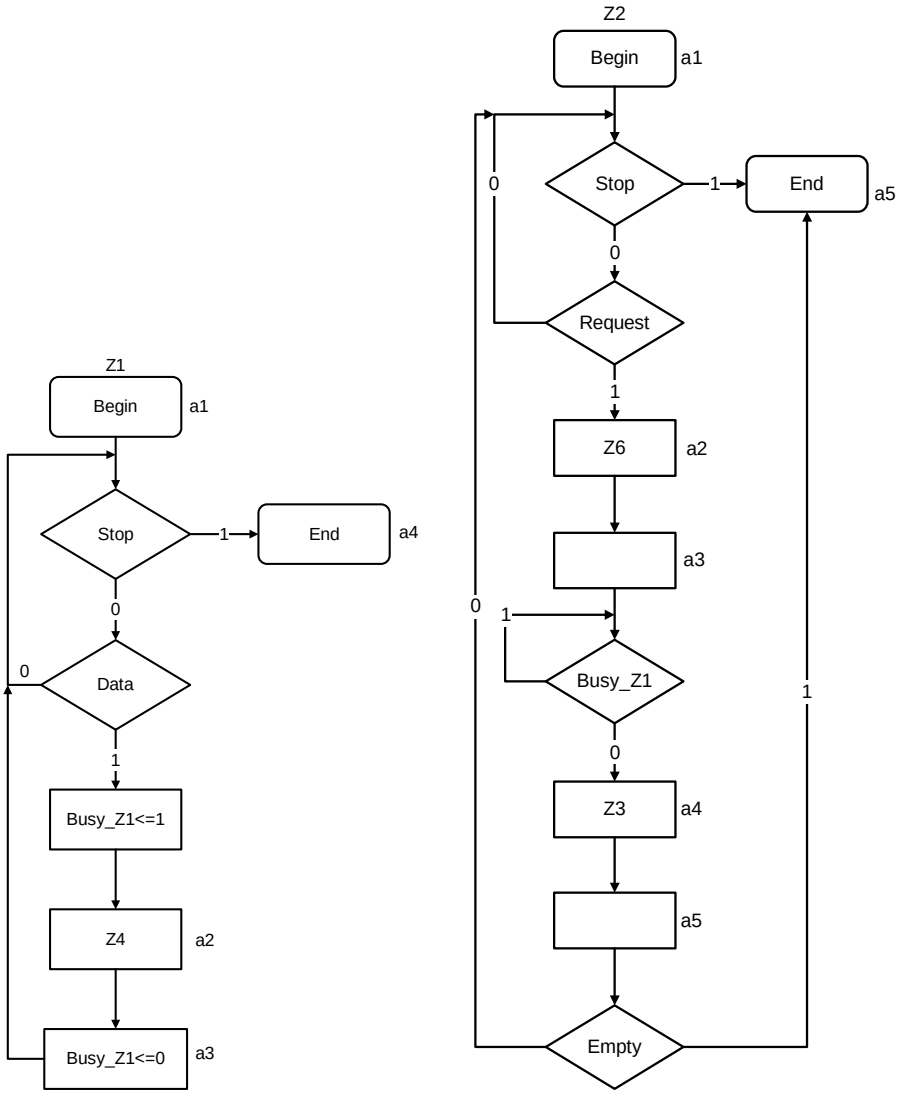


Figura 4.5: Diagramas de fluxo dos módulos Z1,Z2.

paralelo Z1 e Z2,ou seja, em duas *stacks* diferentes (PHFSM), uma para adicionar e outra para retirar nós à árvore respectivamente. Contém também dois sinais de controlo, "*start*" e "*stop*" que permite parar e recomeçar o funcionamento da PHSFM a qualquer momento;

4.1.2 Stack1- Adicionar um novo nó

- **Z1** - Módulo que verifica se existe um novo valor para adicionar à árvore e caso exista activa Z4 para esse efeito;
- **Z4** - Módulo que adiciona um novo nó á arvore caso este valor ainda não exista, percorrendo esta em função do seu valor e colocando este novo nó no seu lugar

respectivo. Inicializa este novo nó e actualiza também o seu nó Pai, ligando-o assim á árvore.

- **Z7** - Módulo activado por Z4 que procura a primeira posição livre na memória para guardar o novo nó que está a ser adicionado á árvore.

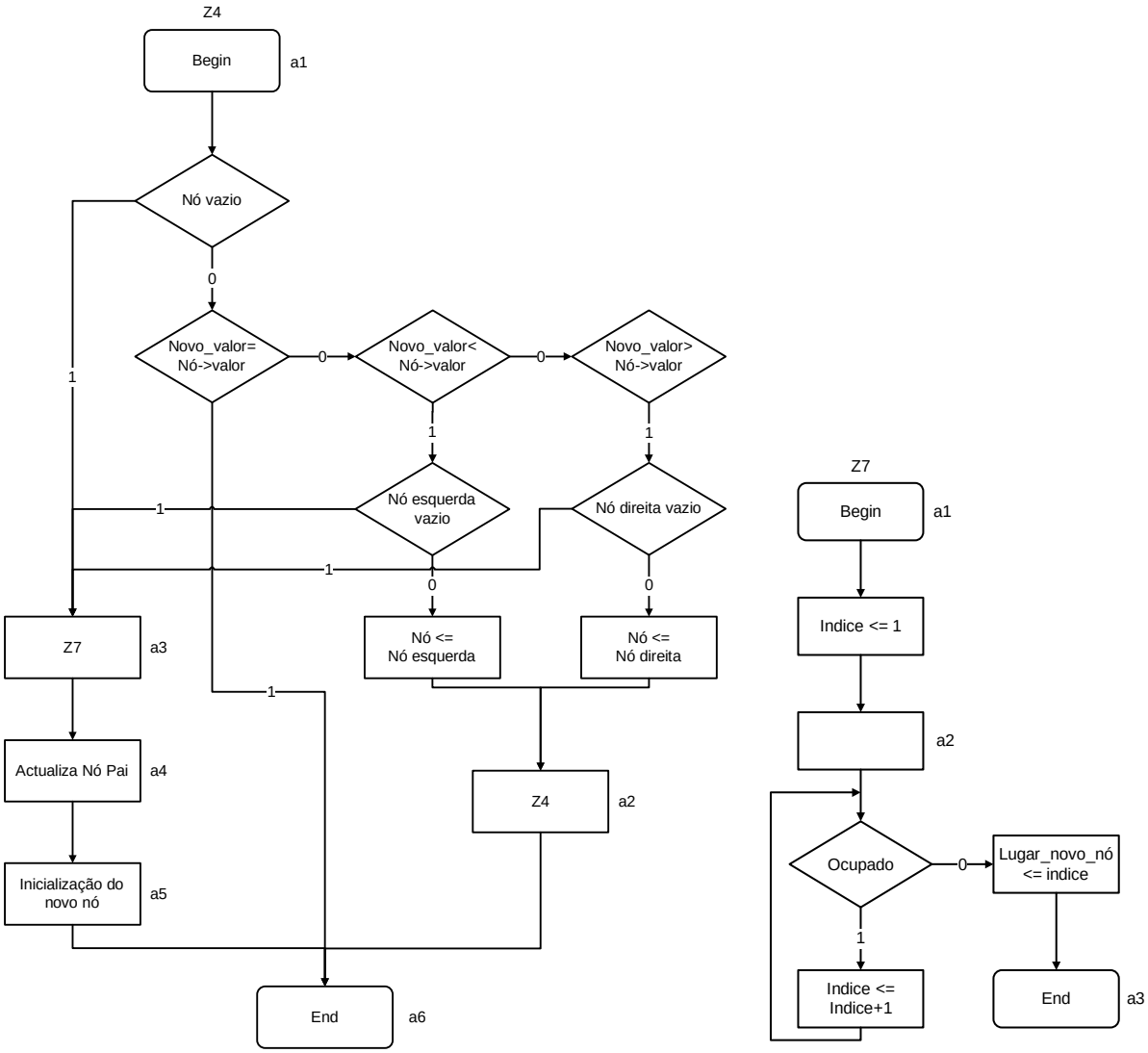


Figura 4.6: Diagramas de fluxo dos módulos Z4 e Z7.

4.1.3 Stack2- Retirar um nó

- **Z2** - Módulo que retira um nó da árvore quando há um pedido, segundo uma regra de prioridade defina em Z6;

- **Z6** - Módulo que define a regra de prioridade do buffer, neste caso percorre toda

a sub-árvore á esquerda até encontrar o nó mais a esquerda, ou seja o valor mais baixo, e envia este valor para a saída do buffer;

- **Z3** - Módulo que fornece sincronização com outros módulos e activa Z5 para a remoção de nós que já foram escolhidos por Z6.
- **Z5** - Módulo que apaga um nó da árvore, previamente escolhido por Z6 e actualiza os nós á sua volta caso existam (nó pai e o nó á sua direita). Quando se trata de apagar o nó raiz também actualiza o valor desse registo.

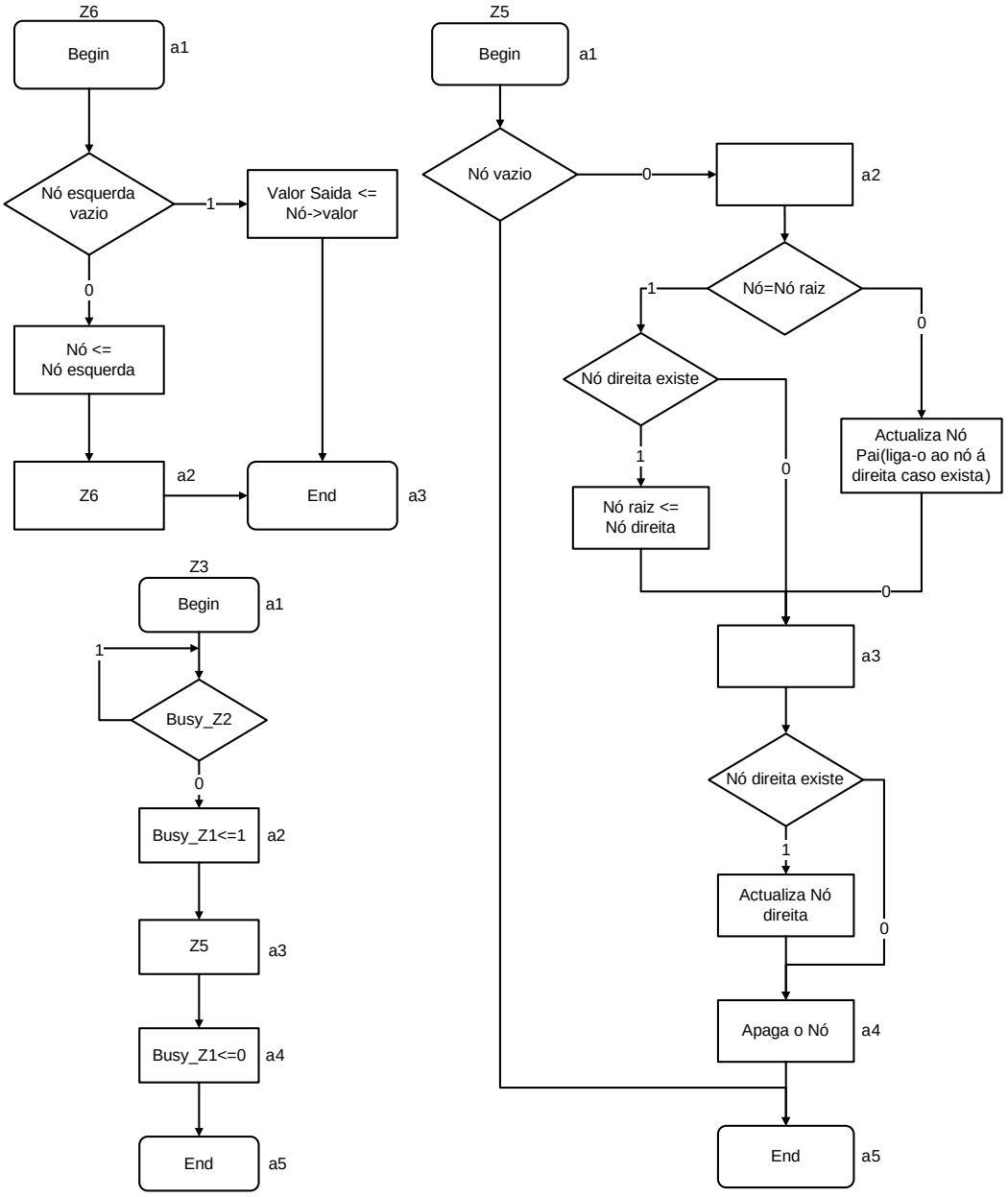


Figura 4.7: Diagramas de fluxo dos módulos Z3, Z5 e Z6.

Este buffer pode facilmente ser adaptado para outro tipo de prioridades consoante a necessidade de outro sistema embutido, através da alteração da regra de prioridade defina em Z6, contudo em alguns casos também pode ser necessário alterar o módulo Z5, uma vez que este foi construído também em função do módulo Z6.

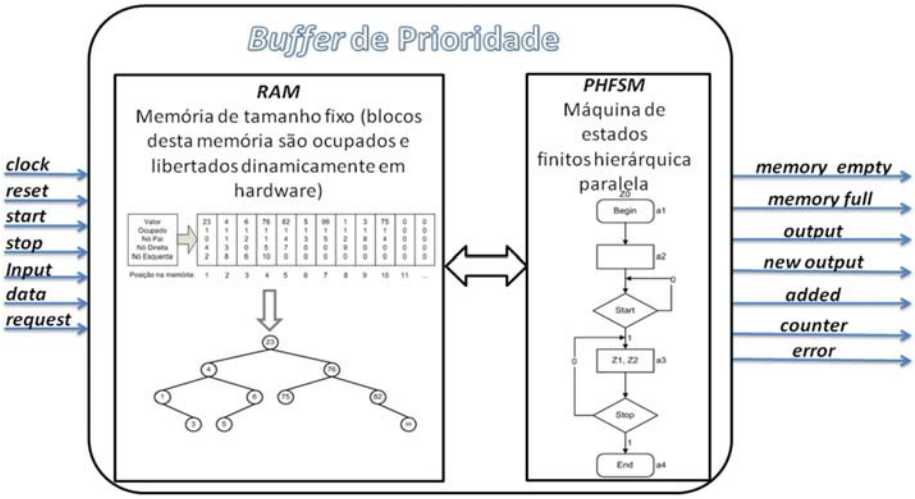


Figura 4.8: *Buffer* de prioridade.

Na figura 4.8 é possível verificar os elementos principais do buffer implementado, já descritos acima, bem como as entradas e saídas deste. Um exemplo da utilização destas entradas e saídas será explicado mais a frente no sistema de controlo do parque de estacionamento onde este *buffer* é utlizado como exemplo.

A memória deste buffer foi inicialmente para teste implementada em memória distribuída, ou seja, foram utilizados os recursos lógicos da FPGA. Deste modo, existe um grande desperdício da capacidade lógica da FPGA pois está a ser usada como memória, como podemos verificar mais á frente nos resultados da síntese. Por isso, faz todo o sentido que esta memória seja implementada em blocos próprios, isto é, em *Block RAMs* que actualmente todas as FPGAs contêm.

A escolha do tipo de memória recaiu numa *True Dual Port*, pois com este tipo de memória é possível ter dois portos com acesso de leitura e de escrita para ambos, bem como ter tamanhos, frequências de relógio e modos de operação diferentes, funcionando assim completamente independentes um do outro. Estas características tornam-se essenciais para a interacção com a árvore binária descrita anteriormente, pois permite ter um porto com o tamanho de um nó inteiro que permita escrever e ler nós inteiros, e outro porto de tamanho mais pequeno, que permita ler e escrever apenas um campo de cada nó.

Para isto utilizou-se um núcleo de propriedade intelectual da Xilinx (Xilinx Logi-

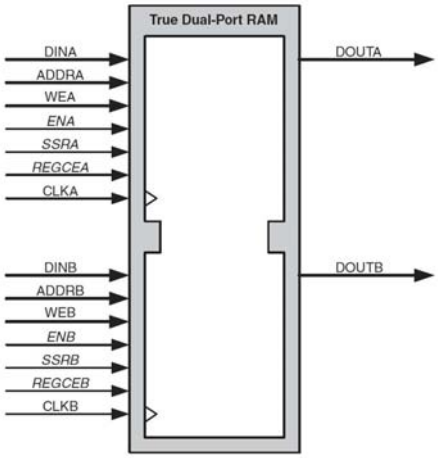


Figura 4.9: *True Dual-port RAM*[21].

CORE™ IP), mais especificamente, o *Block Memory Generator v2.8*, como pode ser visto na figura 4.10.

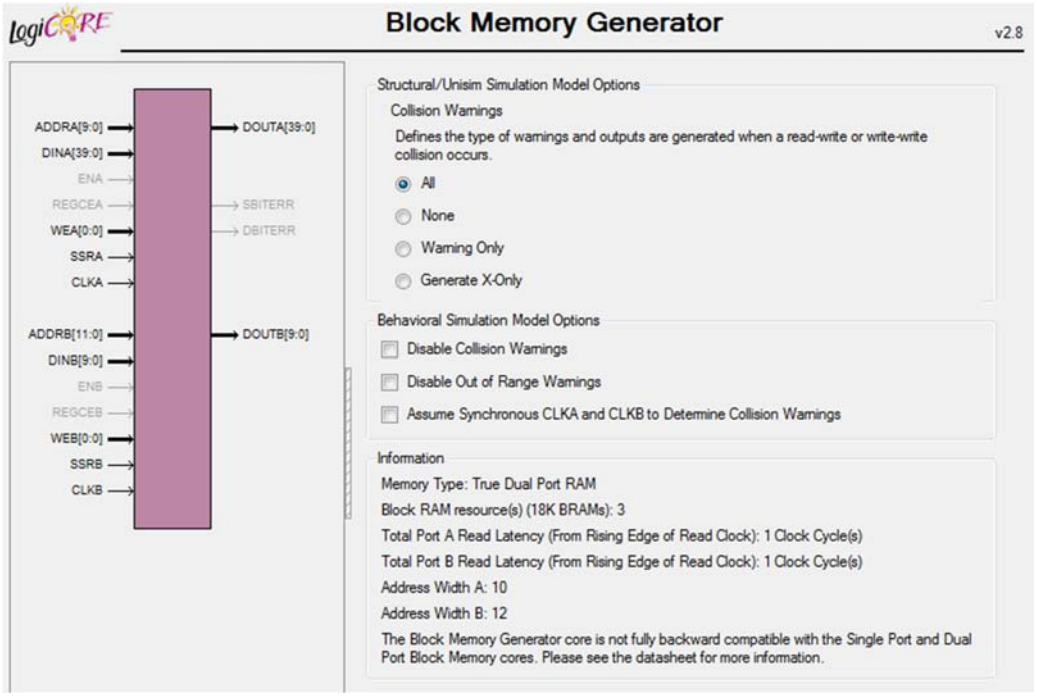


Figura 4.10: Interface e resumo das características da memória utilizada.

Deste modo, o porto A é utilizado para o acesso de um nó inteiro, enquanto que o porto B é utilizado para o acesso de apenas um campo do nó. A memória implementada neste caso tem capacidade para 1024 nós em que cada nó ocupa 40 bits (Porto A), no entanto estes valores podem ser facilmente alterados consoante a aplicação. No caso do sistema do parque de estacionamento este valor já será em princípio suficiente.

O porto B para ser compatível com o porto A deve ter um tamanho múltiplo (2^n). Como cada nó apresenta 5 campos, sendo que no caso do campo *busy* é só um bit, não há necessidade de ocupação de um campo inteiro de vários bits para além, de que com este campo seria necessário dividir o tamanho em memória de um nó por 8 campos. Sendo que nesse caso só se utilizaria praticamente metade dos bits, ou seja, metade da memória reservada para cada nó, pois só 4 campos seriam utilizados mais um bit do campo *busy*.

A condição *busy* pode ser substituída pela simples verificação do bloco de memória reservado para cada nó através do porto A, pois se este estiver todo a zeros, este não está ocupado, uma vez que foi considerado que índice da memória começa no número um. Deste modo, para um melhor aproveitamento da memória, foi eliminado o campo *busy*, ficando o assim cada nó divido nos restantes quatro campos, permitindo assim uma utilização óptima da memória reservada. Assim o porto B terá uma dimensão de 10 bits e uma capacidade de armazenamento 4092 campos correspondentes aos 1024 nós.

4.2 Sistema para teste de *buffers* de prioridade

O *buffer* de prioridade proposto neste projecto permite ser aplicado e numa basta gama de sistemas, pois permite reserva e libertação de memória dinamicamente em *hardware*, sendo deste modo bastante geral. No entanto outras implementações do mesmo *buffer* e mesmo outros tipos de *buffer* de prioridade como uma lista ligada de prioridade podem ser considerados. Para isto, aqui é proposto um sistema que permite testar vários tipos de buffer.

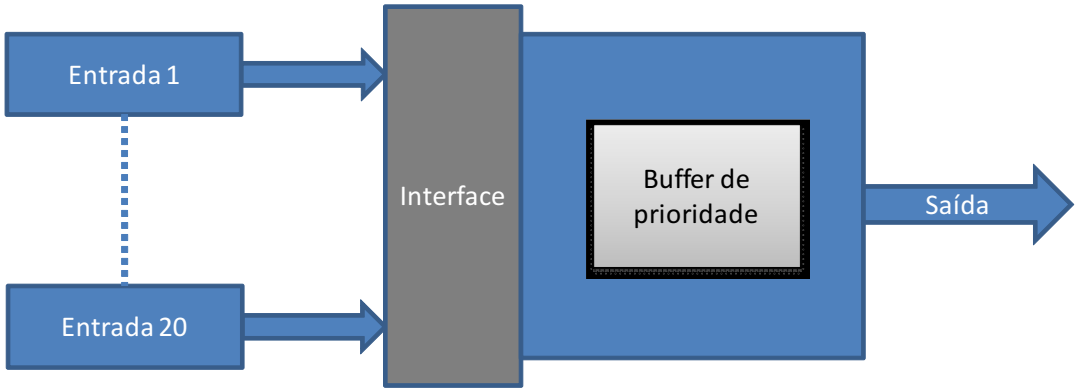


Figura 4.11: Diagrama de blocos do sistema para teste de *buffers* de prioridade.

Na figura 4.11 é possível observar a os blocos principais deste sistema. Para cada uma das 20 entradas deve ser possível alterar a sua frequência e a sua prioridade. A

interface deve seleccionar as entradas segundo uma ordem "*first input first prosessing*", ou seja, enviar para o *buffer* a primeira entrada que chegar. No caso de chegar mais de que uma entrada ao mesmo tempo a selecção é arbitrária. Deste modo o *buffer* de prioridade ligado a esta interface faz a ordenação por prioridades das entradas e fornece á sua saída a respectiva sequência ordenada sempre que este é solicitado.

Assim este sistema permite comparar o desempenho e funcionalidade de diferentes *buffers* de prioridade. (Este também pode ser útil num sistema central que controla um conjunto de parques de estacionamento, onde cada entrada corresponde a um parque de estacionamento.)

4.2.1 Entradas

Assim para a geração de cada entrada foi utilizado um bloco disponível em [48]que utiliza uma das aplicações do algoritmo *linear feedback shift register* (LFSR) como gerador de sequencias "pseudo-aleatórias". A frequência com que este bloco gera novos números e assim novas entradas pode ser alterada pela alteração do relógio fornecido a este bloco. Para isso é utilizado um divisor de frequência dependente do valor da velocidade da entrada respectiva, que pode ser alterado pelo utlizador.

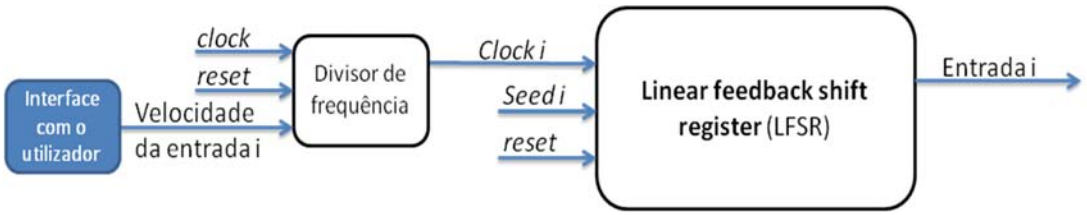


Figura 4.12: Circuito para a geração de cada entrada.

Para cada entrada foi utilizado um circuito como o que se pode observar na figura 4.12, utilizando para isso o construtor da linguagem VHDL *generate*, de modo a implementar vinte circuitos destes em paralelo. Para as diferentes entradas não fornecerem sequências pseudo-aleatórias iguais, a cada entrada foi fornecida uma *seed* diferente.

4.2.2 Interface

Como já foi referido anteriormente, a interface deve enviar para o *buffer* a primeira entrada que chegar ("*first input first prosessing*"), e no caso de chegar mais de que uma entrada ao mesmo tempo a selecção é arbitrária. Para isto, na implementação desta interface considerou-se uma *fifo* de entrada que guarde todas as novas entradas, de modo a fornecer depois ao *buffer* de prioridade. Para a detecção de novas entradas

foi utilizado um ciclo *for* que verifica quando alguma entrada é alterada (nova) e nesse caso activa a posição de um vector construído para esse efeito, que indica as novas entradas.

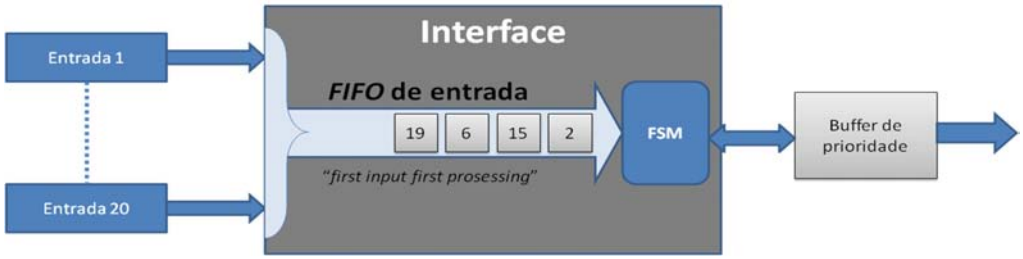


Figura 4.13: Diagrama de blocos da interface.

Assim é utilizada máquina de estados que percorre este vector e quando uma posição deste se encontre a um, a entrada respectiva é adicionada a *fifo* de entrada. Durante estes 20 ciclos de relógio o ciclo *for* anterior volta preencher um novo vector com as novas entradas. Deste modo as entradas novas durante 20 ciclos de relógio são consideradas ao mesmo tempo. O tamanho mínimo desta *fifo* de entrada é igual a vinte, que é no caso em todas as entradas serem alteradas ao mesmo tempo.

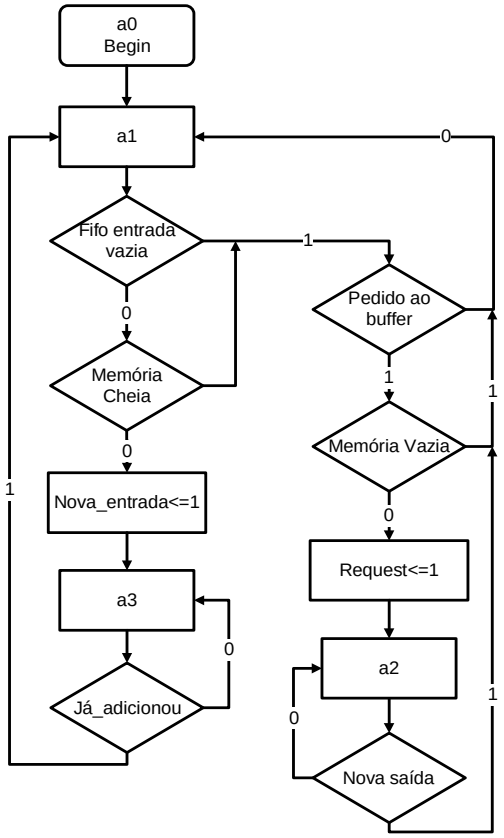


Figura 4.14: Diagrama de fluxo da interface.

A interacção do *buffer* de prioridade com a *fifo* de entrada é feita através de uma simples máquina de estados finitos (FSM) cujo diagrama de fluxo pode ser visto na figura 4.14. Assim esta máquina no estado a1 verifica continuamente o estado da *fifo* de entrada e quando esta contiver alguma entrada nova, ou seja, se não estiver vazia é enviado para o *buffer* a entrada respectiva se este não estiver cheio. Quando isto acontece a máquina passa para o estado a3 onde aguarda que o buffer acabe de adicionar esta nova entrada e só depois disto volta ao estado a1 onde pode voltar a atender a novos pedidos. No estado a1 quando não existe novas entradas e há um pedido ao buffer, neste caso pelo utilizador, se este não estiver vazio, é requisitada uma nova saída segundo uma prioridade. Assim a máquina passa para o estado a2 onde espera que uma nova saída seja disponibilizada pelo *buffer* e só depois volta outra vez para o estado a1. Esta máquina de estados finitos funciona assim como um elemento que fornece exclusão mutua á memória do *buffer*, pois desta maneira não permite que se adicione e retire elementos ao *buffer* ao mesmo tempo.

Para a alteração dos parâmetros das entradas através de um teclado e visualização das saídas num monitor VGA, ou seja, para a interacção com o utilizador são utilizados os blocos disponíveis em [49].

O *buffer* de prioridade descrito anteriormente foi testado com este sistema. Os resultados obtidos mostram um funcionamento correcto por parte deste como era esperado, fornecendo sempre à sua saída a entrada com maior prioridade que a sua memória contenha, no momento em que é feito o pedido (*request*).

4.3 Sistema de controlo para parque de estacionamento

Com este projecto pretende-se a implementação de um sistema de controlo central de um parque de estacionamento utilizando uma FPGA e outra unidade de controlo por carro que controle este e o estacione automaticamente, também implementada numa FPGA. A comunicação entre as unidades de controlo dos carros e o sistema central será efectuada através de uma ligação sem fios, através de interacção remota (por radiofrequência - RF).

No entanto numa primeira fase para simulação e verificação do correcto funcionamento do sistema, as unidades controlo serão implementadas dentro da mesma FPGA. O funcionamento de todo o sistema e algoritmos implementados pode ser verificado através da simulação feita utilizando um monitor VGA. Para interagir com a simulação também deve ser utilizado um teclado PS/2, permitindo assim variar vários

parâmetros da simulação, como por exemplo o número de carros novos por minuto que chegam ao parque de estacionamento.

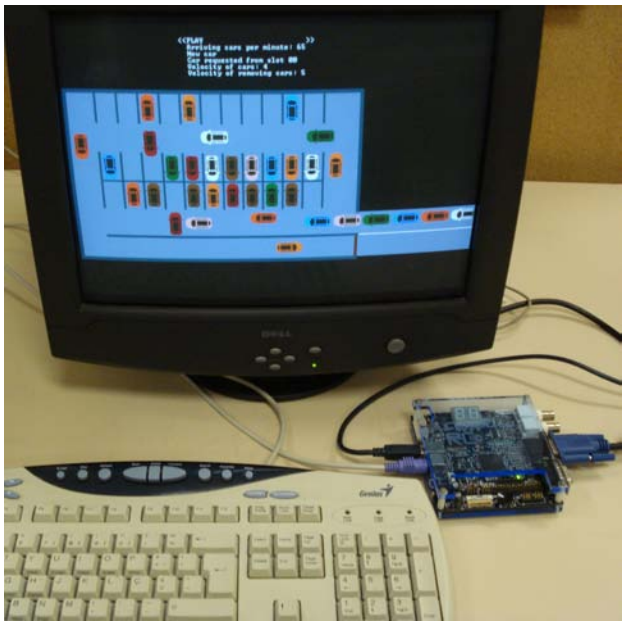


Figura 4.15: Imagem de todo o sistema.

4.3.1 Controlo Central

As funções básicas do sistema de controlo central são:

- Recepção da informação sobre os lugares vazios no parque de estacionamento;
- Implementação de um *buffer* de prioridades de instruções para carros;
- Envio das instruções para os carros;
- Implementação de algoritmos para abrir e fechar as portas de entrada e saída do parque de estacionamento;

O *buffer* de prioridade utilizado aqui (descrito anteriormente), recebe à entrada informação sobre os lugares vazios sequencialmente e fornece na saída o lugar com maior prioridade que se encontra naquele momento vazio, ordenando assim a sequência da saídas sempre que recebe uma nova entrada, colocando-a na sua posição respectiva de prioridade.

Figura 4.16: Funcionamento básico do *buffer* de prioridade.

No caso específico deste sistema, quanto maior for o número do lugar de estacionamento menor será a sua prioridade.

Convém salientar aqui, que este *buffer* de prioridade implementado apresenta mais complexidade e funcionalidade que este sistema requer, pois para este problema em que o número de lugares de um dado parque de estacionamento é conhecido, existem soluções mais simples como já foi referido anteriormente. Deste modo, este tipo de *buffer* apesar de uma maior complexidade pode ser a solução para um maior número de sistemas. Uma aplicação possível é assim o sistema de controlo de um parque de estacionamento que permite também testar o correcto funcionamento deste *buffer* de prioridade de uma forma visual e interactiva.

No caso do parque de estacionamento as entradas e saídas do *buffer* de prioridade da figura 4.8 vão ter as seguintes funções:

- *start*: Esta entrada controla o inicio do funcionamento do *buffer*, neste caso está sempre a '1';
- *stop*: Esta entrada permite parar o funcionamento do *buffer*, neste caso está sempre a '0';
- *input*: Esta é a entrada principal do *buffer*, neste caso serão colocados aqui os números dos lugares de estacionamento que ficam vazios para serem adicionados ao *buffer*.
- *data*: Esta entrada fica a '1' quando um novo lugar ficou vazio, sinalizando assim o *buffer* para este adicionar o lugar que se encontra na entrada anterior "*input* " á memória.
- *request*: Entrada que quando activa, ordena o *buffer* a fornecer o lugar com maior prioridade que se encontra naquele momento disponível na memória.
- *memory empty*: Saída que indica quando a memória está vazia, neste caso indica quando o parque está cheio.

- *memory full*: Saída que indica quando a memória está cheia, neste caso indica quando o parque está vazio.
- *ouput*: Saída principal do *buffer*, neste caso será fornecido aqui o número do lugar de estacionamento com maior prioridade sempre que é feito um pedido ao buffer.
- *added*: Saída que informa quando um novo valor acabou de ser adicionado ao *buffer*.
- *counter*: Saída que informa a quantidade de lugares vazios num determinado momento, através do número de elementos que o *buffer* contém num determinado momento.
- *error*: Saída que informa se há erro na máquina de estados finitos hierárquica paralela, devido ao tamanho da *stack*. Neste caso não será usada, uma vez que o tamanho da *stack* foi dimensionado em função do número de lugares do parque de estacionamento de modo a evitar esta situação.

O controlo central é implementado através de uma máquina de estados finitos (FSM), interagindo assim com praticamente todas as partes do sistema, desde o *buffer* até ao controlo dos carros e das portas, bem como com a interface com que o utilizador controla toda a simulação no monitor VGA.

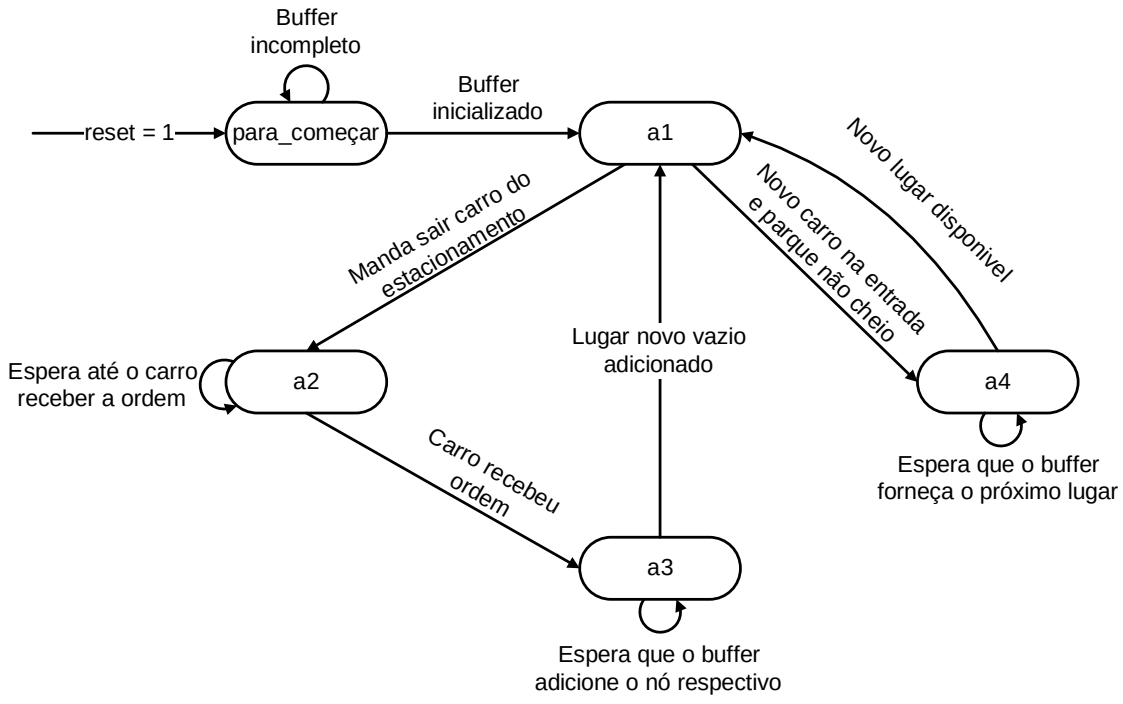


Figura 4.17: Máquina de estados finitos para o controlo central.

Como se pode verificar na figura 4.17, a máquina de estados finitos começa no estado "para_começar", onde inicializa o *buffer*, ou seja, constrói a arvore binária contendo todos os lugares do estacionamento, uma vez que no início o parque está completamente livre. Quando acaba de adicionar todos os lugares do estacionamento e o *buffer* fica cheio a máquina passa para o estado a1, estado no qual a máquina de estados se encontra por defeito, pois se nenhum carro entrar ou sair, a máquina de estados permanecerá neste estado.

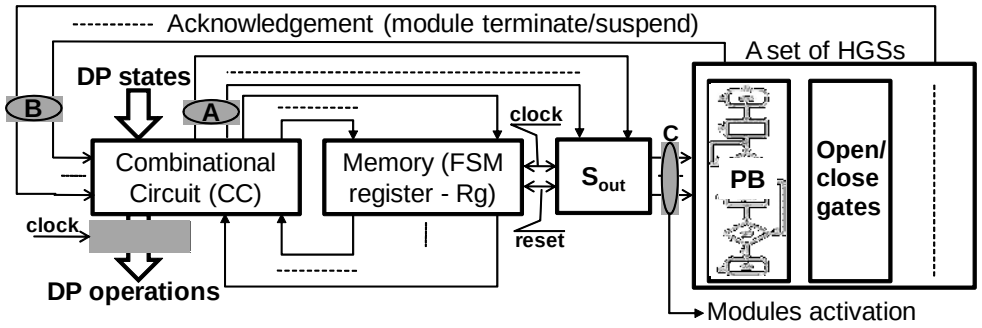


Figura 4.18: Controlo central.

No estado a1 o controlo central está sensível às ordens "aleatórias" de saída de carros geradas por um bloco externo, bem como às ordens do utilizador através da interface criada para esse efeito. Assim, quando é recebida uma ordem para um carro sair do estacionamento, a máquina passa para o estado a2 e espera nesse estado que o carro receba a ordem de saída. Quando a ordem é recebida pelo carro, este envia um aviso (*"acknowledge"*) ao controlo central a informar que recebeu a ordem, passando a máquina de estados para o estado a3 onde é adicionado ao *buffer* um novo nó com o valor de prioridade do respectivo lugar. Assim que este acabe de ser actualizado com o novo lugar vazio a máquina volta para o seu estado por defeito (a1), podendo atender a novos pedidos.

Quando chega um novo carro á porta de entrada, se houver lugares livres disponíveis, ou seja, se o parque não estiver cheio, o controlo central passa directamente do estado a1 para o estado a4, aonde vai esperar que o *buffer* forneça o lugar vazio com maior prioridade que existe no momento em que este foi requisitado; caso o parque esteja cheio, espera até um lugar fique livre para passar para o estado a4. Quando este fornece o novo lugar, a máquina de estados volta outra vez para o seu estado por defeito (a1) e é enviado para o carro que solicitou o lugar no qual ele deve estacionar.

O estado a2 é utilizado como elemento sincronizador entre o controlo central e o controlo de carros, esperando assim que o carro receba a ordem de saída. Os estados a3 e o a4 funcionam como elementos sincronizadores do *buffer*, evitando assim o acesso

à memória ao mesmo tempo para apagar e remover nós, evitando assim erros que caso contrário podiam originar, como por exemplo ficarem nós na memória desligados e isolados do resto da árvore binária.

O controlo central através de um módulo disponível em [48], utiliza uma das aplicações do algoritmo linear feedback shift register (LFSR) [3] como gerador de saídas de carros "pseudo-aleatórias", em que a frequência de novas saídas de carros "aleatórias" pode ser alterada pelo utilizador através da interface criada para esse efeito. A variação desta velocidade de saída e obtida através da variação do *clock* fornecido ao módulo que gera as saídas "pseudo-aleatórias".

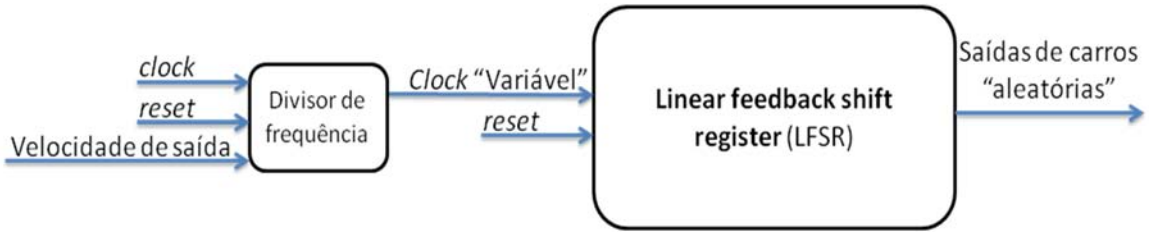


Figura 4.19: Gerador de saídas de carros "aleatórias".

Para a chegada de um determinado número de carros por minuto, que pode ser alterado pelo utilizador durante a simulação, foi também necessária a implementação de um módulo para fazer a divisão, uma vez que esta não é directamente sintetizável para este caso (em que um dos operandos não é constante); assim é determinado o intervalo de tempo entre cada carro novo. Para esta função também foi necessária a utilização de um divisor de frequência para tornar mais simples a contagem dos segundos de intervalo entre os carros novos.

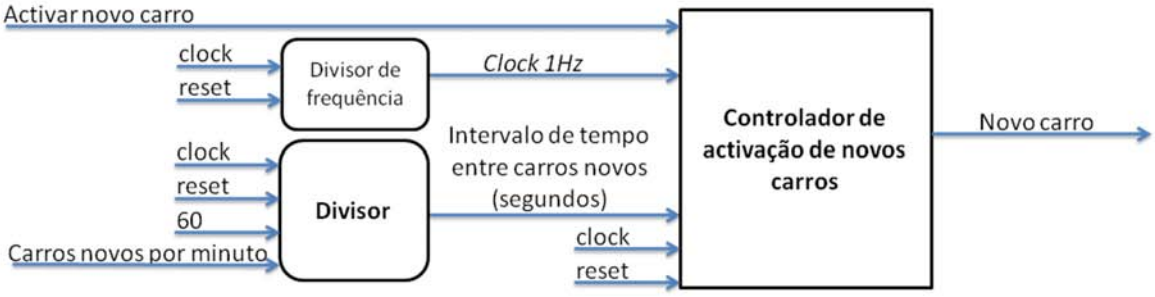


Figura 4.20: Activação de novos carros.

Controlador das portas

Para o controlo das portas de entrada e de saída considerou-se que não há a necessidade da porta fechar depois de um carro passar quando já tem outro carro pronto para passar

por ela, tornando assim o sistema mais rápido e eficiente, evitando assim paragens desnecessárias aos carros. Deste modo, para a sua implementação foi utilizado um simples contador para cada porta, um para a entrada e outro para a saída, em que cada contador controla a sua porta respectiva; Estes contadores são só incrementados quando um carro chega a porta respectiva e decrementados apenas quando um carro passa completamente pela porta respectiva. Assim, se o contador for maior que zero, a porta respectiva é aberta, caso contrário, a porta é fechada.

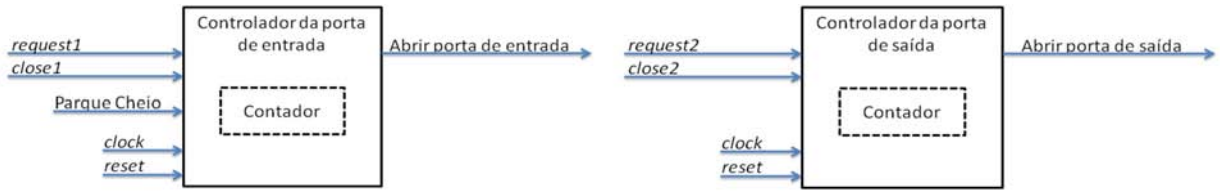


Figura 4.21: Controlo das portas de entrada e saída.

O controlo da porta de entrada também tem em atenção quando o parque se encontra cheio, fechando esta porta mesmo que um carro novo chegue, só voltando abrir quando pelo menos um lugar ficar livre.

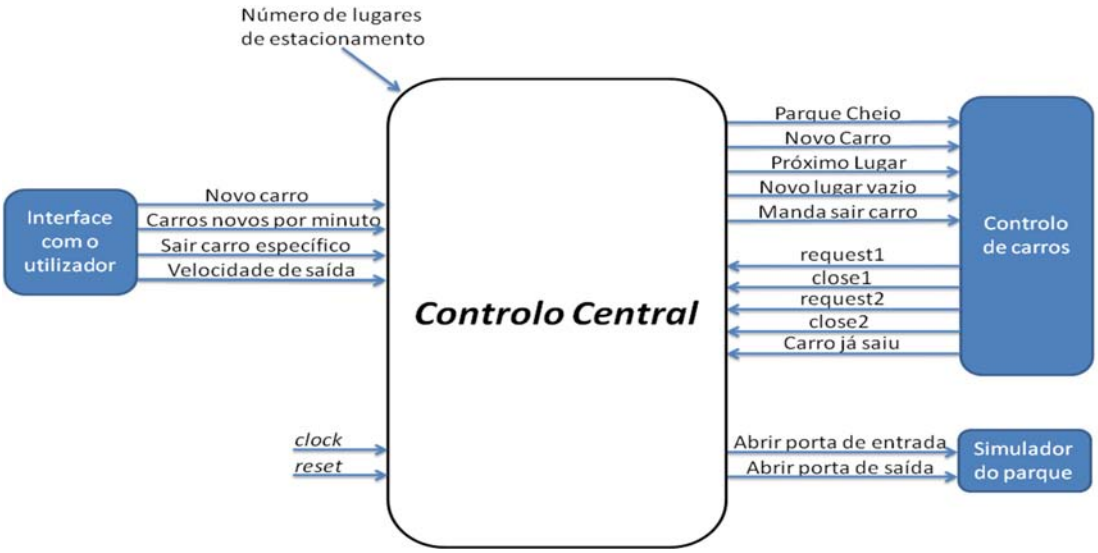


Figura 4.22: Interface do controlo central com o sistema.

4.3.2 Controlo dos carros

4.3.2.1 Sensores

Para evitar colisões entre os carros houve a necessidade de criar um sistema que indicasse para cada carro se este podia andar sem correr o risco de colidir com outro carro

qualquer. Para este caso foi utilizado uma máquina de estados finitos (FSM) em vez de um ciclo for, gastando assim menos recursos á FPGA, uma vez que há tempo suficiente para percorrer todos os carros através da máquina de estados entre cada uma actualização de posições. Esta máquina compara a posição de cada carro com a de todos os outros continuamente, e assim actualiza constantemente um vector de tamanho igual ao número de carros, que informa cada carro se este pode continuar a andar ou não.

Deste modo, é possível que quando dois carros se dirigem para o mesmo espaço se bloquem mutuamente, ou seja, um para de andar por causa do outro e vice-versa. Para evitar esta situação foi necessário dar prioridades diferentes em função da direcção e sentido de cada carro, para que um carro pare antes de bloquear o outro.

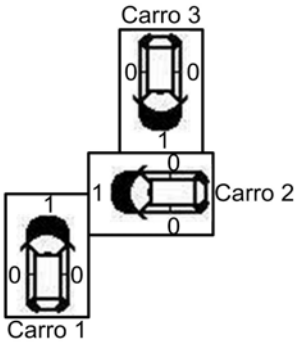


Figura 4.23: Exemplo de prioridade no controlo de carros.

Os valores que se encontram á volta dos carros na figura 4.23 referem-se ao alcance que um carro consegue ver outro no sentido do respectivo número. Assim como se pode verificar na figura, o carro 2 quando se desloca para a esquerda tem maior alcance (1) que o carro 1 no seu lado direito (0), parando assim o carro 2 antes de entrar na zona do carro 1, evitando o bloqueio deste também. Desta maneira o carro 1 nunca vai detectar o carro2, seguindo primeiro o carro 1 e depois o carro 2 quando o caminho ficar livre, sem colisões ou bloqueios de ambos os carros como é pretendido. O que acontece entre o carro 2 e o carro 3 é exactamente o mesmo que foi descrito para a situação anterior entre o carro 1 e 2, sendo que o carro 3 vai ser o ultimo a voltar a andar, pois vai ter de esperar pelo carro 2 e este pelo carro1.

Quando o carro se desloca para a direita, devido à forma como foi desenhado o parque, não há risco de haver colisões com outros carros noutros sentidos. Deste modo, não foi necessário alterar o seu alcance nesse sentido.

Para além disto, também foi necessário considerar o caso de quando dois carros deslocam em diferentes direcções e chegam exactamente ao mesmo tempo a um determinado ponto como se pode ver na figura 4.24.

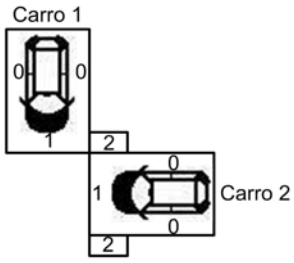


Figura 4.24: Prioridade quando os carros chegam ao mesmo tempo.

Para evitar esta situação de bloqueio de ambos os carros foi necessário criar um excepção para este caso específico, onde se aumentou o alcance lateral (para 2) dos carros que se deslocam na horizontal para a esquerda (carro2 da figura 4.24) permitindo assim, estes detectar primeiro os outros carros que se deslocam na vertical, dando assim prioridade a estes quando os dois carros chegam ao mesmo tempo. O alcance lateral dos carros que se deslocam para esquerda só foi alterado mesmo nessa situação, pois caso contrário com este alcance para todo o comprimento do carro, este iria bloquear em situações que não devia, como por exemplo no caso do carro2 por causa do carro 3 no exemplo da figura 4.23.

Neste módulo além de criar um vector que indica a cada carro se este pode andar, também é criado um vector de igual forma para os carros saberem quando podem mudar de direcção, pois para este movimento, estes precisam mais espaço; senão quando mudassem de direcção podiam colidir ou bloquear com outros carros.

Na vida real para implementar este sistema na prática este módulo poderia ser substituído por sistemas de controlo já presentes alguns veículos modernos como por exemplo o *Adaptative cruise control* (ACC) [50].

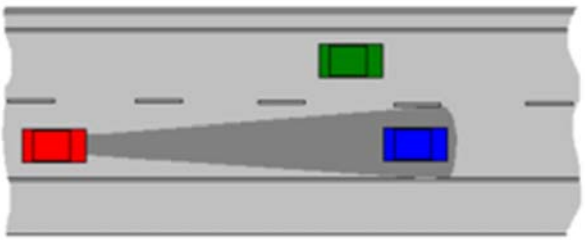


Figura 4.25: Diagrama de controlo inteligente de carros (ACC).

4.3.2.2 Controlo dos carros

Para cada carro está associada a seguinte informação:

car_x_reg: Posição do carro na horizontal em relação ao lado esquerdo do ecra;

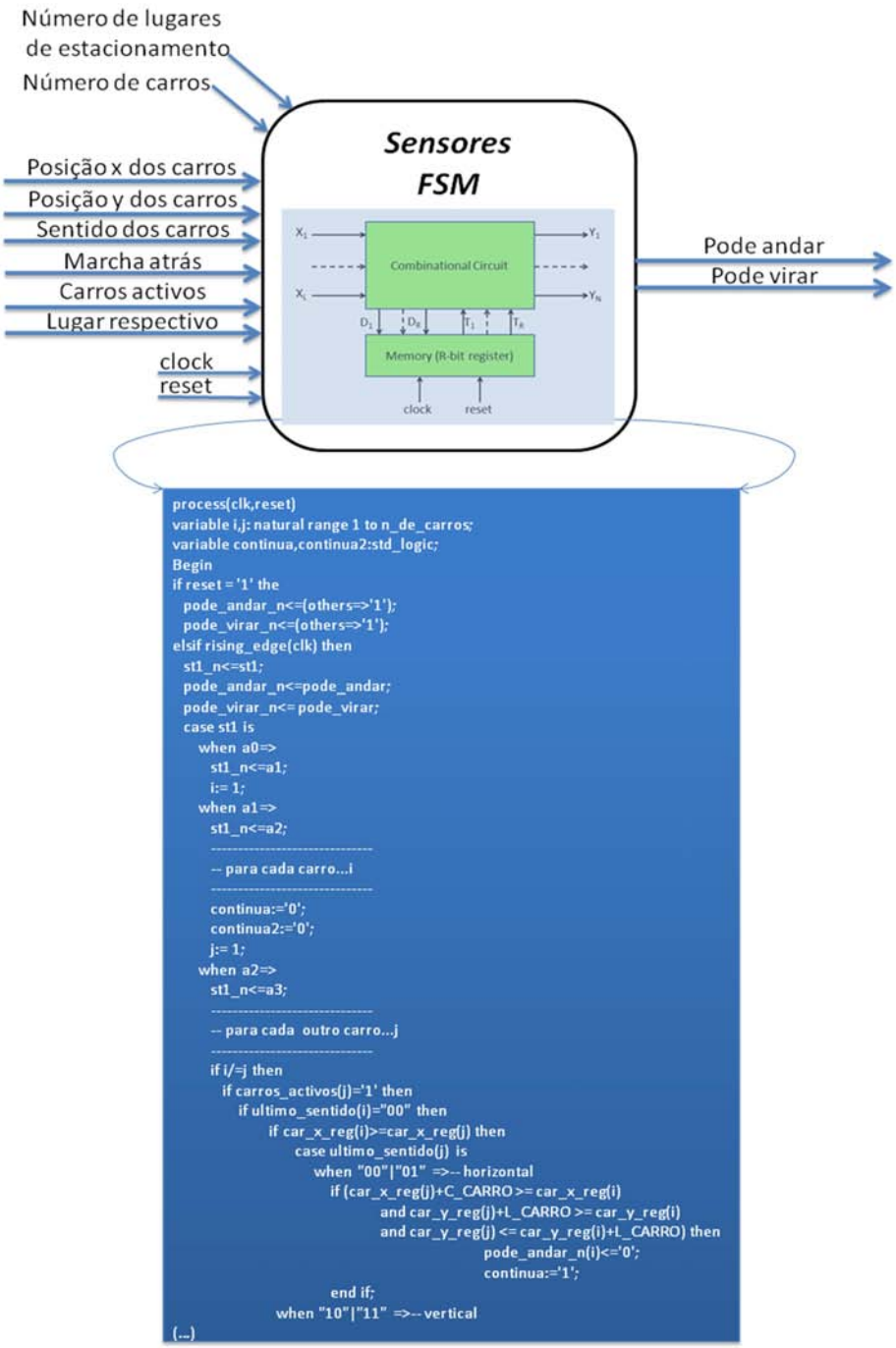


Figura 4.26: Sensores.

car_y_reg: Posição do carro na vertical em relação ao topo do ecra;

ultimo_sentido: Direção e sentido actual do carro;

etapa: Etapa em que o carro se encontra;

lugar_resp: Lugar onde o carro deve estacionar;

carros_activos: Indica se o carro está activo;

marcha_atras: Indica se o carro está a andar em marcha atrás;

Para o controlo dos carros propriamente dito, foram definidas 23 etapas por onde estes vão passar:

- Etapa 1: Nesta etapa encontram-se todos os carros inactivos. Aqui os carros não estão visíveis no ecrã, pois a sua posição passa o lado direito do ecrã visível. Quando o controlo central manda activar mais que um carro, so um destes em cada ciclo de actualização de todos os carros e que activa e passa para a etapa 2, pois se activasse mais do que um eles ficavam bloqueados mutuamente;
- Etapa 3: Quando chega aqui o carro faz um pedido ao controlo central para abrir a porta de entrada e um lugar para estacionar;
- Etapa 4: Aqui o carro espera que a porta de entrada fique aberta para ele poder entrar no parque;
- Etapa 5: O carro entra no parque e recebe do controlo central o lugar no qual ele deve estacionar, quando este passa completamente pela porta sinaliza o controlo central que por ele já pode fechar a porta de entrada. O lugar é recebido quando o carro está a passar pela porta, pois assim dá ao buffer tempo mais do que suficiente para disponibilizar o novo lugar bem como não corre o risco de outro carro já ter feito outro pedido ao buffer.
- Etapa 6: Aqui o carro verifica se o lugar já está livre, pois em situações de muito tráfego o carro anterior que estava nesse lugar pode ainda não ter saído completamente do lugar devido á continua passagem de carros. Também aqui os carros passam para diferentes etapas consoante o lugar de estacionamento que lhes foi atribuído.
- Etapa 13: Quando o carro chega a esta etapa, este encontra-se estacionado no seu lugar de estacionamento e ficará nesta etapa até o controlo central o mande sair passando para a próxima etapa.

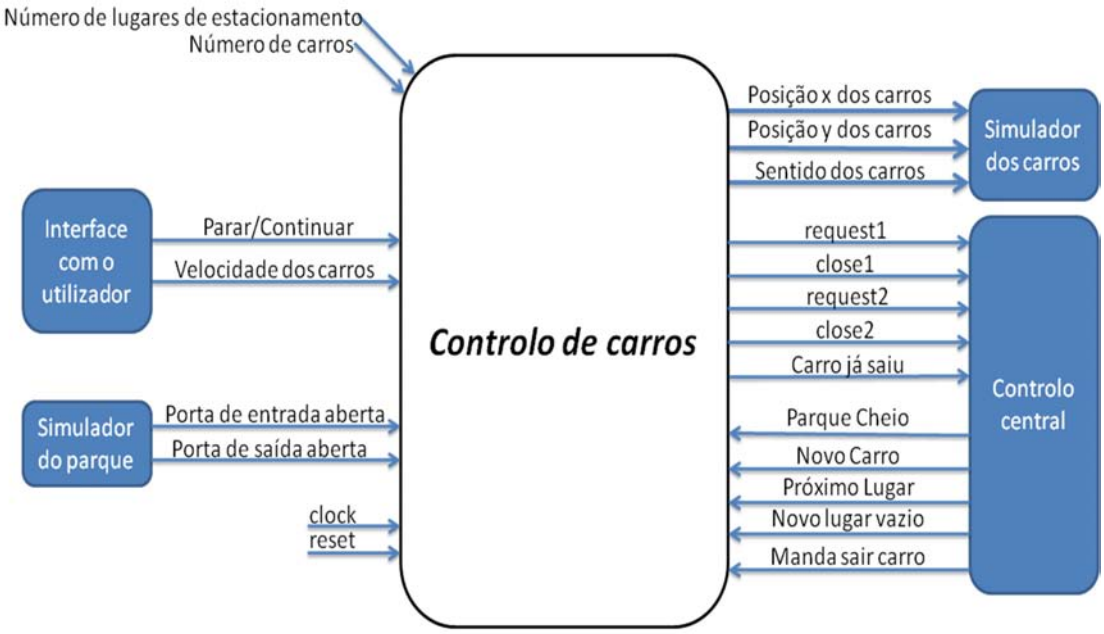


Figura 4.27: Interface do controlo de carros com o sistema.

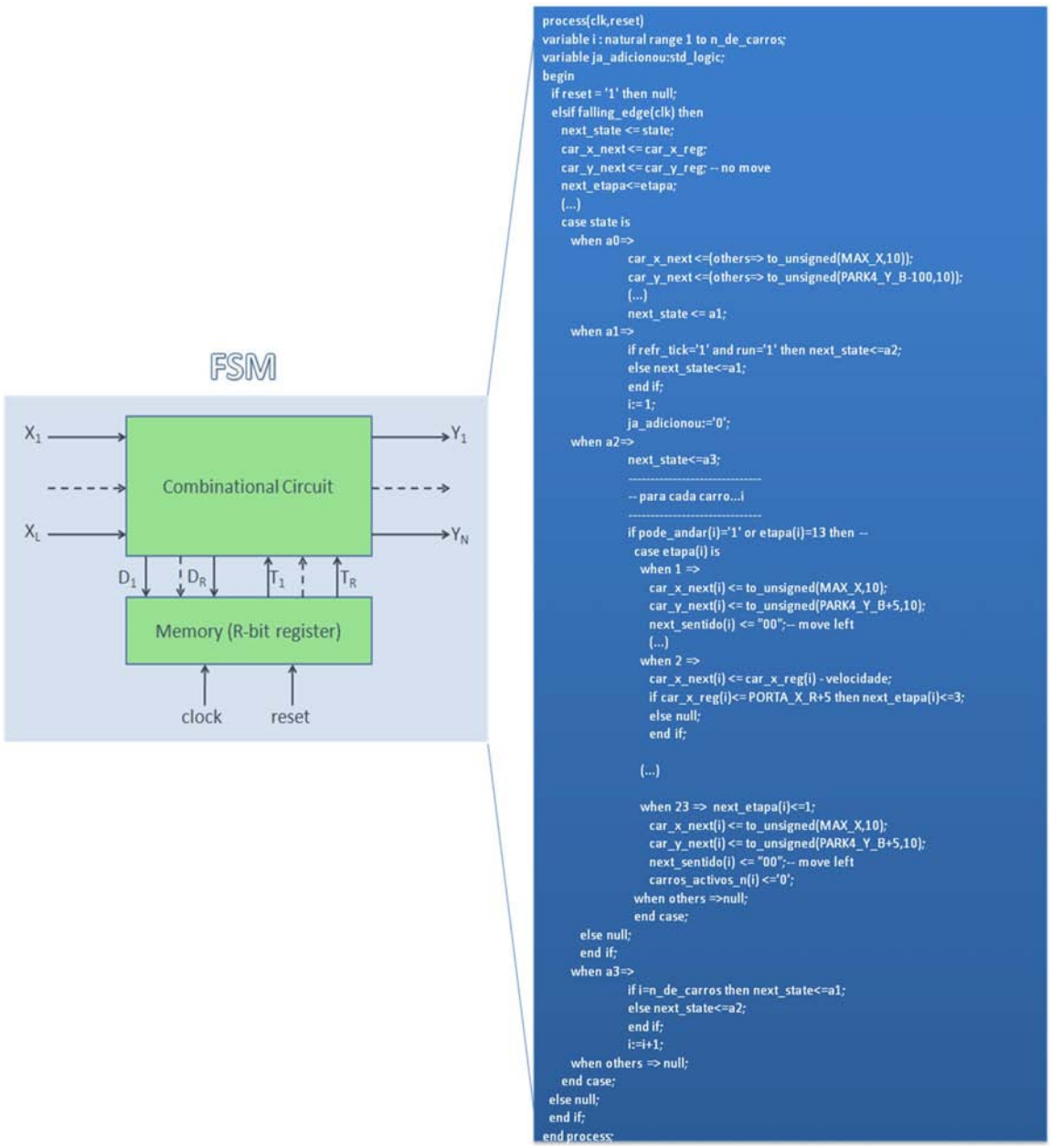
- Etapa 19: Quando o carro chega a esta etapa o carro faz um pedido ao controlo central para abrir a porta de saída, e passa para a etapa 20;

- Etapa 20:Nesta etapa o carro espera até a porta de saída estar aberta para poder passar;

- Etapa 21: Aqui quando o carro passa completamente pela porta de saída avisa o controlo central que por ele já pode fechar a porta de saída.

- Etapa 23: Quando o carro atinge esta etapa, este já saiu da parte visível do ecrã. Passa assim para a etapa 1 ficando inactivo. Deste modo pode voltar a ser utilizado pelo controlo central, assim que este o voltar a activar.

No resto das etapas o carro movimenta-se normalmente sempre que possa andar ou virar de direcção consoante a etapa. Na situação dos carros da 2ª e 3ª fila quererem sair ao mesmo tempo para evitar bloqueios mútuos foi dada prioridade aos carros da 2ª fila, pois estão em lugares com maior prioridade. Assim os carros que estão estacionados 3ª fila só saem do estacionamento quando a estrada por perto estiver livre.




```
process(clk,reset)
variable i : natural range 1 to n_de_carros;
variable ja_adicionou:std_logic;
begin
  if reset = '1' then null;
  elsif falling_edge(clk) then
    next_state <= state;
    car_x_next <= car_x_reg;
    car_y_next <= car_y_reg; -- no move
    next_etapa<=etapa;
    (...)
    case state is
      when a0=>
                car_x_next <=(others=> to_unsigned(MAX_X,10));
                car_y_next <=(others=> to_unsigned(PARK4_Y_B-100,10));
                (...)
                next_state <= a1;
    when a1=>
                if refr_tick='1' and run='1' then next_state<=a2;
                else next_state<=a1;
                end if;
                i:= 1;
                ja_adicionou:='0';
    when a2=>
                next_state<=a3;
                ----------------------------
                -- para cada carro...i
                ----------------------------
                if pode_andar(i)='1' or etapa(i)=13 then --
                 case etapa(i) is
                  when 1 =>
                    car_x_next(i) <= to_unsigned(MAX_X,10);
                    car_y_next(i) <= to_unsigned(PARK4_Y_B+5,10);
                    next_sentido(i) <= "00";-- move left
                    (...)
                  when 2 =>
                    car_x_next(i) <= car_x_reg(i) - velocidade;
                    if car_x_reg(i)<= PORTA_X_R+5 then next_etapa(i)<=3;
                    else null;
                    end if;

                  (...)

                  when 23 => next_etapa(i)<=1;
                    car_x_next(i) <= to_unsigned(MAX_X,10);
                    car_y_next(i) <= to_unsigned(PARK4_Y_B+5,10);
                    next_sentido(i) <= "00";-- move left
                    carros_activos_n(i) <='0';
                  when others =>null;
                  end case;
      else null;
      end if;
    when a3=>
                if i=n_de_carros then next_state<=a1;
                else next_state<=a2;
                end if;
                i:=i+1;
    when others => null;
    end case;
  else null;
  end if;
end process;
```

Figura 4.28: Máquina de estados finitos para o controlo de carros.

O controlo dos carros também e efectuado através de uma simples maquina de estados finitos (FSM), que percorre todos carros e actualiza a posição de cada um em função da sua etapa, em vez do uso de um ciclo for, pelos mesmos motivos já referidos acima. O número de actualizações de posições dos carros por cada "*refresh*" do monitor permite variar velocidade dos carros, que pode ser alterada durante a simulação pelo utilizador através da interface criada para esse efeito.

Para este sistema funcionar o intervalo tempo entre cada actualização de posições dos carros tem que ser no mínimo o tempo necessário para que todos os carros possam

comparar as suas posições uns com os outros para saberem se podem andar ou não, ou seja, é necessário um tempo mínimo para que as máquinas de estados finitos percorram todos os carros antes da próxima actualização de posições. Neste caso, isto verifica-se para a velocidade máxima implementada.

4.3.3 Simulação no monitor VGA

4.3.3.1 Simulador dos carros

Para apresentar um simples objecto no monitor VGA foi utilizado como base o exemplo apresentado em [27]. Neste caso, para representar um carro foi utilizado duas ROMs para este apresentar duas cores, ou seja, uma ROM para cada cor, ficando assim com uma melhor apresentação. Como o carro pode estar na horizontal como na vertical, foi necessário assim duas ROMs para cada direcção. Em relação aos diferentes sentidos, não foi necessário adicionar novas ROMs, pois para mudar o sentido da imagem basta mudar ordem de leitura das ROMs respectivas, mudando assim o sentido do carro.

Deste modo, ao todo foram utilizadas 4 ROMs para apresentar o carro com duas cores, em todas as direcções e sentidos. Para a construção das ROMs, primeiro o carro foi desenhado num simples editor de imagem, ficando as imagens com o seguinte aspecto:

Figura 4.29: Imagens para representar os carros.

Para processar estas imagens para ROMs foi feito um pequeno *script* em Matlab que gera uma matriz de uns e zeros em função do nível de cor de cada pixel.

```
imread('imagem_do_carro.jpg');
ans1=ans(:,:,1)>195;% para ver
imshow(ans1)
ans1=ans(:,:,1)<195;% para copiar
ans1 = cast(ans1,'uint8');
```

Figura 4.30: *Script* Matlab para conversão da imagem numa matriz de bits.

A leitura destas imagens foi feita sincronamente, para que a sua implementação seja feita em *Block Rams*, de maneira a fazer uma melhor gestão dos recursos da FPGA utilizada. Devido a necessidade de simular um número mínimo e aceitável de carros para

se poder verificar o funcionamento do parque de estacionamento, foi experimentado simular 40 carros iguais ao referido acima, todos em paralelo. No entanto verificou-se que deste modo, só a simulação dos carros ocupava os recursos todos da FPGA. Assim foi necessário utilizar outra maneira de os simular, de modo a que seja feita uma utilização mais eficiente dos recursos da FPGA e que seja possível a simulação e controlo de todo o parque de estacionamento.

Para isso, em vez de serem geradas entidades em paralelo para cada carro, foi utilizada só uma entidade que verifica se num dado pixel existe algum carro, e nesse caso coloca os valores desse carro no resto da estrutura que simula os carros.

Para esse efeito, foi criado um processo sensível ao pixel x, onde sempre que este seja actualizado, o processo através de um ciclo for, compara as posições de todos os carros com o próximo pixel, para quando este for actualizado, se este existir algum carro, os valores deste já estejam colocados nos registos que simulam o carro e prontos para apresentar este devidamente no monitor VGA.

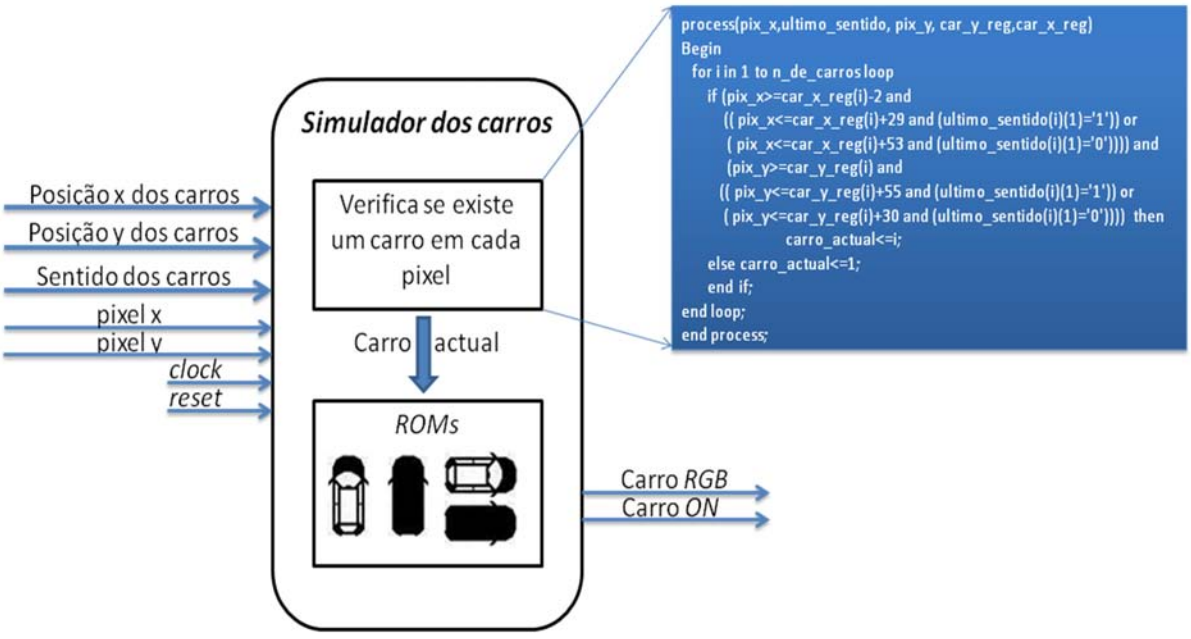


Figura 4.31: Simulador dos carros.

Devido a necessidade de um bom tempo de resposta neste caso, foi necessário a utilização de um ciclo *for*, em vez de uma máquina de estados finitos que ocupa menos recursos da FPGA. Deste modo, verificou-se uma grande redução dos recursos usados pela FPGA para esta simulação, como era pretendido, no entanto também observou-se que para um número de carros maior que 20 é necessário dois processos em paralelo, para permitir actualizar a tempo o carro do actual num dado pixel. Pois, quando só

se utiliza um processo, o atraso dos circuitos para os carros com índice maior que 20, torna-se demasiado grande e observava-se pequenos erros gráficos no monitor VGA.

4.3.3.2 Simulador do parque

Nesta parte, foram geradas todas as outras partes gráficas do parque, como as paredes deste, divisões dos lugares de estacionamento, portas entrada e saída do parque e a estrada (chão). A quantidade de divisões para os carros, bem como o próprio tamanho do parque vai ser gerado em função do número de lugares do parque de estacionamento, que é um parâmetro genérico de entrada. Por motivos de geometria deste parque o número de lugares deve ser múltiplo de 3, uma vez que existem 3 filas de estacionamento dentro do parque.

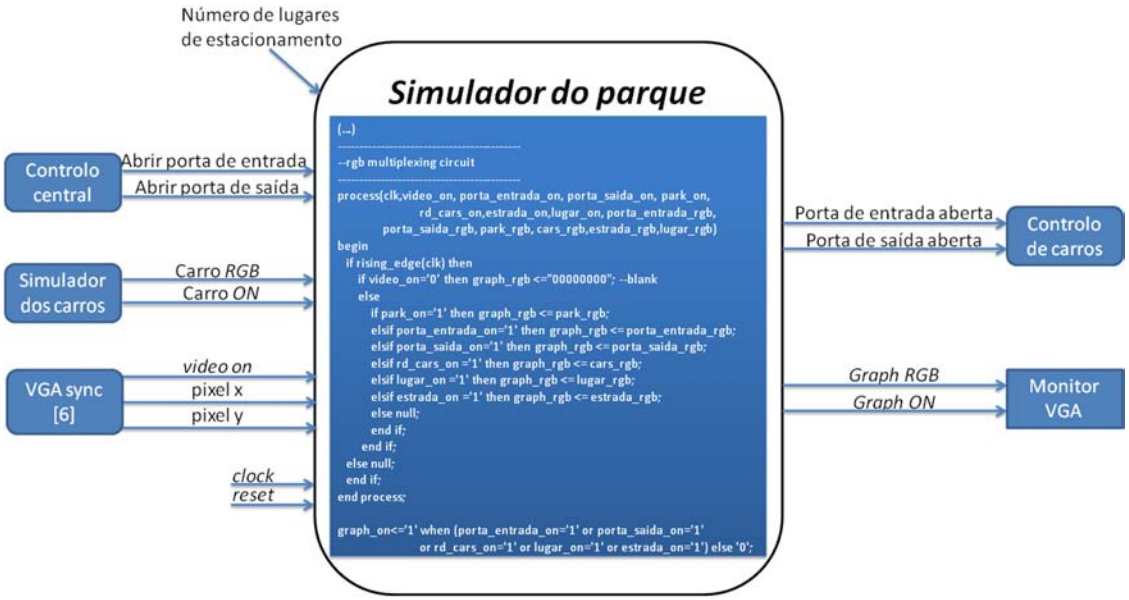


Figura 4.32: Simulador do parque.

Neste módulo também é efectuado alteração do estado das portas por ordem do controlo central, bem como a sinalização para os carros quando as portas estão abertas ou fechadas.

Na figura 4.33 pode-se observar como consiste a geração de toda a parte gráfica da simulação, onde cada circuito gerador em resumo faz:

1. Utiliza as coordenadas do pixel_x e pixel_y para comparar com a posição dos elementos gráficos pelo qual este é responsável;

2. Caso algum dos elementos gráficos deste gerador estejam na região das coordenadas actuais, este coloca a '1' o sinal ON a indicar que este elemento encontra-se

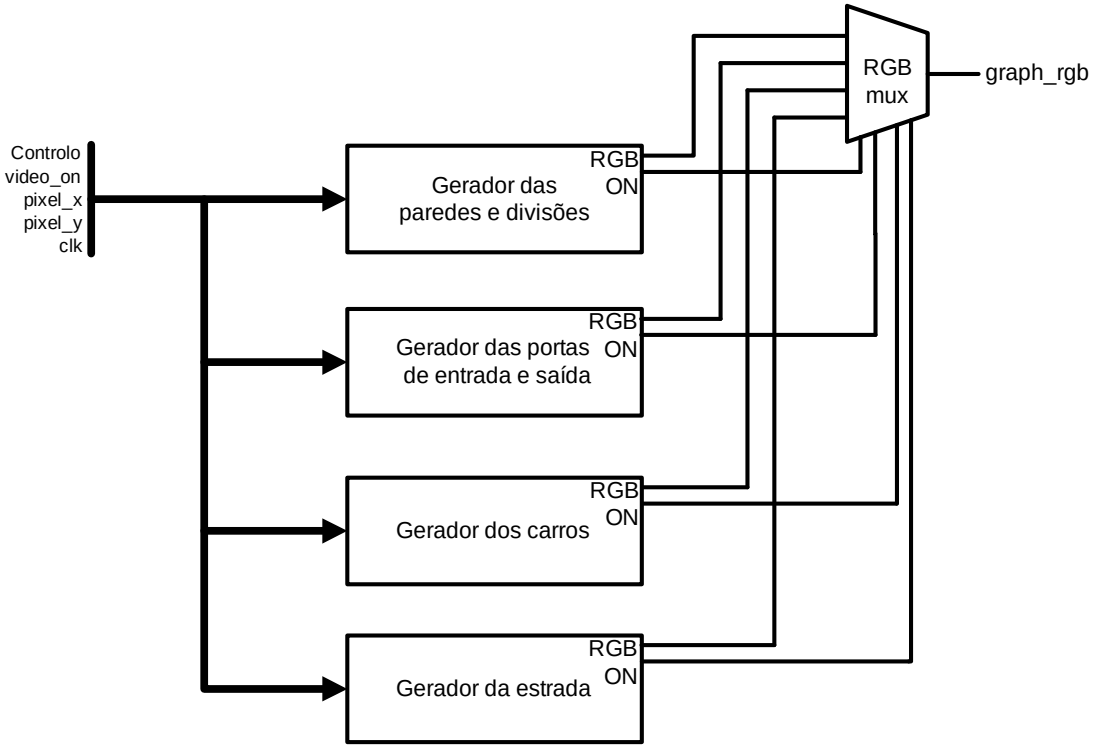


Figura 4.33: Diagrama de fluxo para a geração da parte gráfica da simulação.

activo;

3. Através do sinal RGB, este especifica a cor do elemento que se encontra activo. Como na mesma região do ecrã pode haver mais que um elemento activo (como por exemplo o carro e a estrada), a selecção do elemento que fica visível e efectuada pelo circuito RGB *mux*, que escolhe a cor do elemento activo com maior prioridade. Naturalmente neste caso foi dada maior prioridade aos carros em relação a estrada.

4.3.4 Interface com o utilizador

Neste módulo, foram aproveitadas a funcionalidades oferecidas pelos blocos fornecidos em [49], utilizando as interfaces para teclado e para o monitor VGA, permitindo assim escrever texto no monitor VGA. Deste modo, foi criado um menu no qual permite ao utilizador através do teclado controlar a simulação do parque de estacionamento. Este menu pode ser percorrido de uma linha para as outras através da setas para cima e para baixo do teclado. As opções que este menu oferece são:

1ªLinha - "PLAY/PAUSE" : Quando a tecla ENTER é premida nesta linha, esta opção permite a qualquer momento parar os carros ou pô-los a andar consoante estes

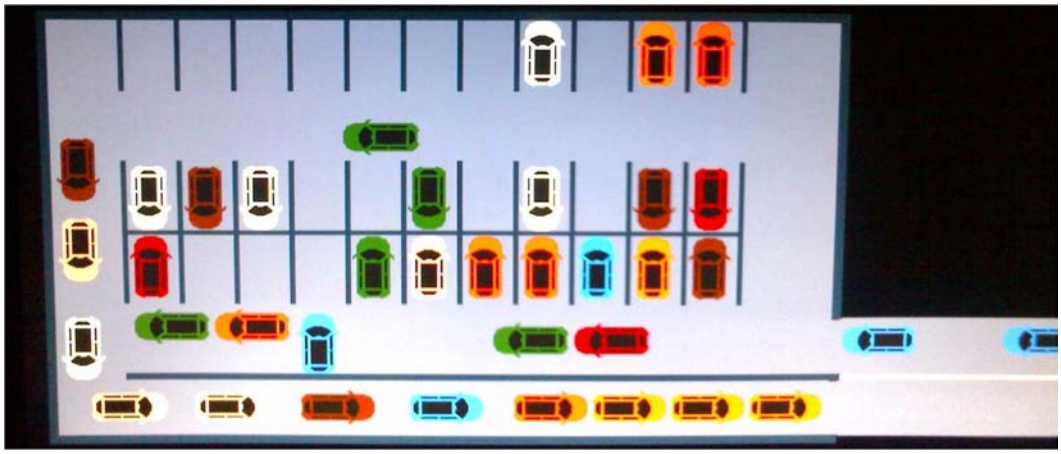


Figura 4.34: Parte gráfica do parque de estacionamento.

estejam parados ou em movimento;

2ªLinha - "Arriving cars per minute: –": Consoante os números introduzidos no teclado nesta linha, o controlo central vai activar carros sucessivamente entre intervalos de tempo, cujo valor é determinado em função do número presente nesta linha. Este intervalo de tempo é sempre actualizado quando se altera este número, de maneira a que se obtenha uma aproximação do número de carros novos por minuto pretendida;

3ªLinha - "New car": Quando a tecla ENTER é premida nesta linha, um novo carro é activado, passando da etapa 1 para a seguinte. Se no momento em que esta opção é seleccionada não houver nenhum carro na etapa 1, o próximo carro que chegar a esta etapa é activado imediatamente;

4ªLinha - "Car requested from slot: –": Esta opção permite ao utilizador, consoante os números introduzidos no teclado, mandar sair o carro que se encontra estacionado no respectivo lugar;

5ªLinha- "Velocity of cars: -": Aqui é possível alterar a velocidade com que os carros se movimentam na simulação. Esta velocidade varia entre 0 a 9 actualizações de posição por cada *refresh* do monitor VGA.

6ªLinha-"Velocity of removing cars: -": Esta opção também permite ao utilizador variar esta velocidade entre 0 e 9, permitindo assim variar a frequência com que os carros saem dos lugares de estacionamento. Esta alteração é feita através da variação do relógio fornecido ao módulo que gera as saídas "pseudo-aleatórias". Naturalmente quando este número esta a zero, este módulo pára de gerar saídas de carros automaticamente.

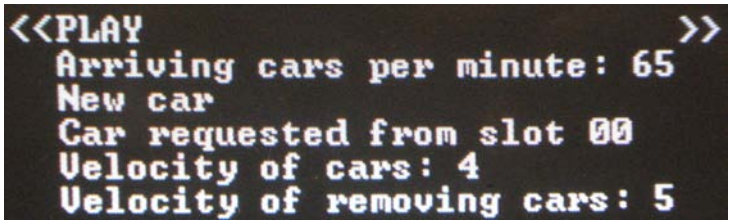


Figura 4.35: Imagem do menu.

4.3.5 Parque de estacionamento – "*Top level*"

Neste módulo é feita a ligação de todos os blocos descritos anteriormente, bem como a sincronização entre estes.

Devido ao facto de o controlo de carros implicar circuitos relativamente longos, aumentando assim o atraso total destes circuitos, foi necessário para o correcto funcionamento do controlo dos carros, a diminuição da frequência do relógio fornecido a este módulo. Por razões de sincronismo, utilizou-se também este relógio de frequência mais baixa para o módulo do controlo central. Assim para os módulos de controlo central e controlo dos carros foi utilizado um relógio com uma frequência de 6,25 MHz. No resto dos módulos foi utilizado o *clock* por defeito fornecido pelo cristal da placa usada (50 MHz).

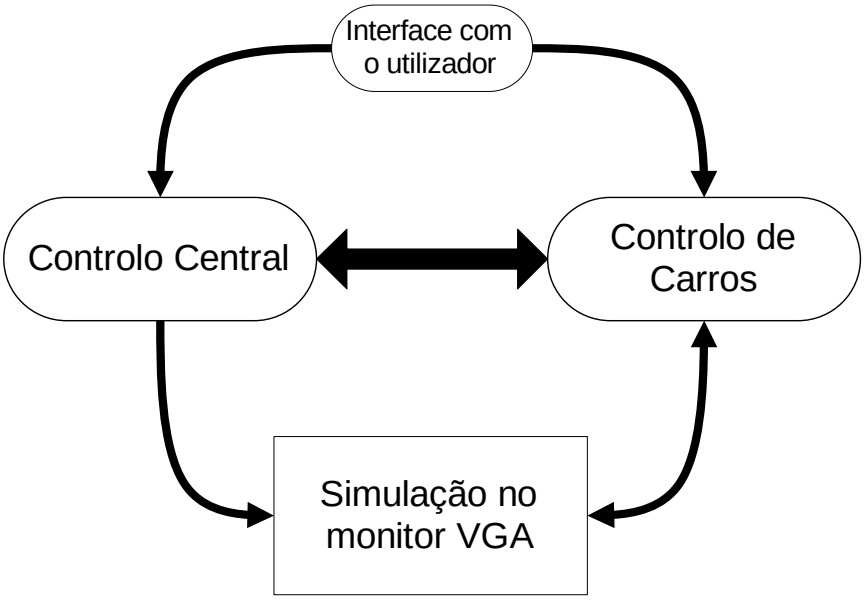


Figura 4.36: Diagrama de blocos geral.

4.4 Resultados

O buffer foi primeiro testado com sucesso através do ModelSim e depois sintetizado e implementado em FPGAs da família Xilinx Spartan 3, obtendo-se os resultados de acordo com o que era esperado. No desenvolvimento deste trabalho percorreu-se as seguintes etapas:

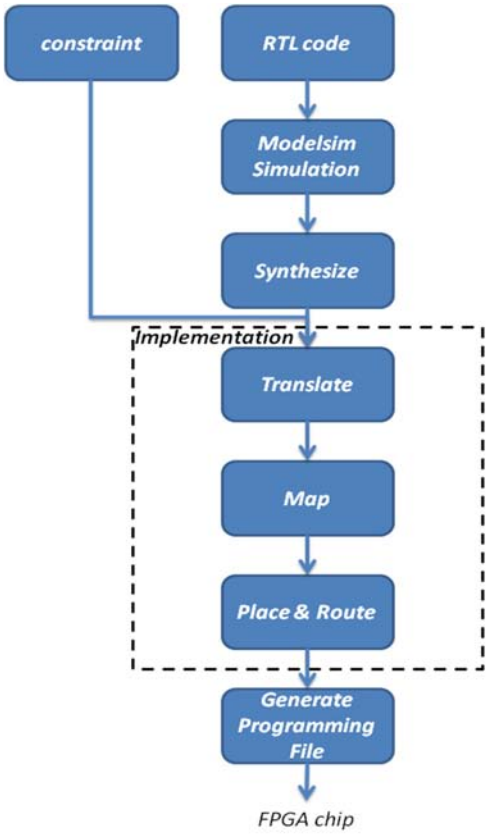


Figura 4.37: Fluxo de desenvolvimento do projecto.

O sistema para teste de *buffers* de prioridade também permitiu o teste deste *buffer* e assim permitiu inclusive o seu próprio teste. Ambos funcionam como era esperado. Para implementação do sistema do parque de estacionamento, foi necessário utilizar a placa RC10 da Celoxica uma vez que esta contém uma FPGA com três vezes mais capacidade do que a FPGA da placa Nexys 2, pois este sistema ocupa bastante recursos físicos.

Para 33 lugares de estacionamento e para 38 carros simulados, com a memória do *buffer* de prioridade implementada em memória distribuída obteve-se os seguintes resultados:

Recursos	**Ocupados**	**Disponíveis**	**Utilização**
Slices	12,526	13,312	94%
LUTs	21,219	26,624	79%
IOBs	27	221	12%
Block RAMs	16	32	50%
DCMs	1	4	25%

Tabela 4.1: Utilização geral da FPGA (*buffer* em memória distribuída).

Módulo	**LUTs**	***Block* RAMs**
Parque de estacionamento	71 - 0.3%	-
Controlo Central	489 - 1.8%	-
Buffer de prioridade	6308 - 23.7%	-
Controlo dos Carros	5089 - 19.1%	-
Sensores	2253 - 8.5%	-
Simulador dos carros	5151 - 19.4%	8 - 25%
Simulador do Parque	307 - 1.1%	-
Interface com o utlizador	487 - 1.8%	-

Tabela 4.2: Utilização de cada módulo (*buffer* em memória distribuída).

Para o mesmo sistema, mas com a memória do *buffer* de prioridade implementada desta vez em block RAMs obteve-se os seguintes resultados:

Recursos	**Ocupados**	**Disponíveis**	**Utilização**
Slices	10,053	13,312	75%
LUTs	16,410	26,624	61%
IOBs	35	221	15%
Block RAMs	19	32	59%
DCMs	1	4	25%

Tabela 4.3: Utilização geral da FPGA (*buffer* em *Block* RAMs).

Módulo	**LUTs**	***Block* RAMs**
Parque de estacionamento	69 - 0.3%	-
Controlo Central	363 - 1.4%	-
Buffer de prioridade	1283 - 4.8%	3 - 9.4%
Controlo dos Carros	5089 - 19.1%	-
Sensores	2253 - 8.5%	-
Simulador dos carros	5151 - 19.4%	8 - 25%
Simulador do Parque	307 - 1.1%	-
Interface com o utlizador	487 - 1.8%	-

Tabela 4.4: Utilização de cada módulo (*buffer* em *Block* RAMs).

O controlo do parque e dos carros, bem como a interacção com o utilizador funciona correctamente e como era esperado. Verifica-se também como era esperado uma grande redução dos recursos lógicos da FPGA, com a utilização de Block RAMs para a implementação da memória, passando de 94% para 75% de *slices* ocupados. É possível observar esta diferença em mais pormenor através da utilização do número de LUTs do *buffer* em ambos os casos, que passa de 6308 para 1283. Convêm salientar ainda que no último caso a memória tem capacidade para 1024 lugares enquanto que no primeiro caso só tinha para 38 lugares, ou seja, quanto maior for o tamanho da memória necessária maior será esta diferença.

Capítulo 5

Interacção remota

Sumário

Neste capítulo é feita a descrição da implementação prática em FPGAs do módulo que permite a interface com o *transceiver* RF escolhido para a respectiva comunicação sem fios, numa linguagem VHDL. Para isto, é descrita uma solução que utiliza uma decomposição hierárquica do tipo "*bottom-up*" desde a camada mais mais a baixo que gera o relógio (SCLK) para o protocolo SPI, que faz a comunicação com o *transceiver* RF, até a camada mais a cima "*top-level*" que recebe e envia os dados que o sistema precisa de transferir para outro remotamente (sem fios). Deste modo é feita uma decomposição do problema em "sub-problemas" mais pequenos, simplificando assim a solução final. No final é implementada e testada a interacção remota no sistema do parque de estacionamento do capítulo anterior.

5.1 Implementação

A ligação entre a FPGA e *transceiver* CC1101 é efectuada através de uma interface SPI de quatro fios (SI, SO, SCLK e CSn) onde a FPGA funciona como *Master* e o *transceiver* como *Slave*.

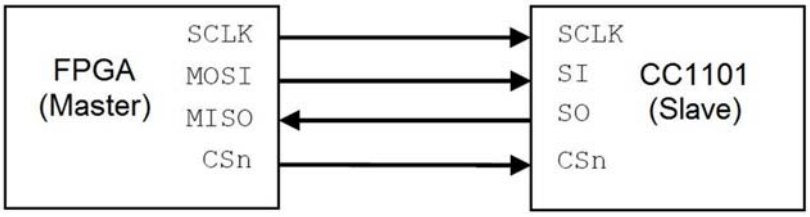


Figura 5.1: Interface SPI.

5.1.1 Acessos SPI

Todas as transferências nesta interface SPI são efectuadas de forma a enviar primeiro o bit mais significativo (MSB). Todas estas transferências começam através de um *header* byte [22] enviado pelo *master* para o *slave*.

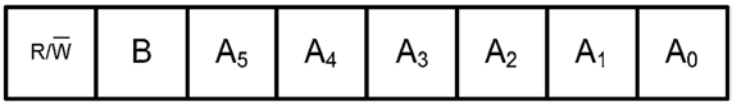


Figura 5.2: Estrutura do *header* byte[22].

Este é composto como se pode ver na figura 5.2, por:

- 1 bit $R/\overline{W}$: quando é uma operação de escrita é igual a 0 e na leitura é igual a 1;
- 1 bit B: indica se um simples acesso quando é igual a 0 ou em *burst* quando é igual a 1;
- 6 bits de endereço ($A5$-$A0$): define o endereço a que o *master* quer aceder para ler ou escrever.

Sempre que o *master* escreve algum byte na interface SPI, o *transceiver* envia também um byte de informação (*status* byte) sobre o seu estado através da linha MISO. Este byte contém informação muito útil ao *master* (ver anexo A). Através do seu primeiro bit, o CHIP_RDYn, sabe quando o *transceiver* está pronto para comunicar (só quando este está a zero). Os restantes bits permitem ao *master* saber o estado interno do *transceiver* e os bytes que o *transceiver* contém nas *fifos* consoante seja uma escrita ou leitura.

Esta interface pode ser utilizada para vários tipos de comunicação com o *transceiver*. Além da leitura e escrita de um simples registo, o *transceiver* também pode ser acedido em modo *burst*, ou seja, é possível ler ou escrever vários registos em sequência sem intervalo. Outro acesso muito importante nesta interface com o *transceiver*, são os comandos *strobe*, pois permitem controlar o *transceiver* directamente. Para uma correcta comunicação com o *transceiver*, alguns tempos de espera tem de ser respeitados como se pode ver na figura 5.3.

Num acesso simples, o bit B (*burst*) naturalmente tem de ser igual a zero. Assim depois da transmissão do respectivo *header* byte é transmitido ou lido um byte consoante o bit $R/\overline{W}$. Depois disto, pode acontecer mais um acesso simples mantendo nesse caso o sinal CSn a zero, caso contrário este sinal volta a um até o próximo acesso.

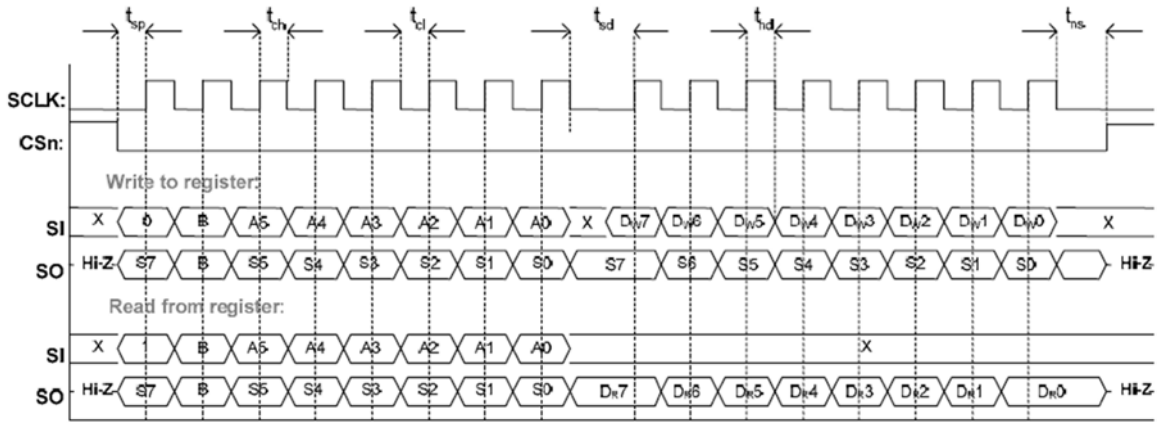


Figura 5.3: Exemplo de uma escrita e uma leitura de um registo.

Num acesso em burst, é enviado um *header* byte seguido de bytes consecutivos para serem escritos ou lidos consoante o bit $R/\overline{W}$ até o sinal CSn voltar a um, acabando assim este acesso.

Este tipo de acessos é bastante útil, primeiro para configurar o *transceiver* num modo pretendido e depois para ler e escrever das respectivas *fifos*, para a comunicação sem fios propriamente dita. O *transceiver* tem 47 registos que permitem configurar este em diferentes modos de comunicação sem fios (como mudar a modulação, o canal, etc). Estes registos encontram-se no intervalo de endereços entre o 0x00 e o 0x2E. Além destes registos, é preciso também ter em conta o registo do endereço 0x3E, pois através deste é possível controlar a potência de saída do *transceiver*.

Para aceder às duas *fifos* é utilizado o mesmo endereço, o 0x3F. Para aceder á *fifo* de transmissão o bit $R/\overline{W}$ do *header* byte deve ser igual a zero. Caso contrário é a *fifo* de recepção que é acedida. Deste modo, na *fifo* de transmissão só se pode escrever e na *fifo* de recepção só se pode ler.

Outro tipo de acesso a interface SPI, são os comandos *Strobe*. Estes podem ser vistos como acessos de um só byte a determinados registos, ou seja, só é enviado o *header* byte, não havendo transferência de mais nenhum byte. Quando estes comandos são enviados para o *transceiver*, este vai proceder a diferentes operações específicas consoante a ordem enviada pelo *master*. Estes comandos podem ser usados várias funções como activar o modo de transmissão, activar o modo de recepção, fazer reset ao *transceiver*, etc (todos os comandos *strobe* podem ser consulatdos no anexo A).

Os endereços para estes comandos estão no intervalo entre 0x30 e 0x3D. Esta gama de endereços tem dupla função, pois podem ser usados como comandos *strobe* ou podem ser usados para aceder a registos só de leitura, que contém várias informações importantes acerca do estado corrente do *transceiver*. Convém referir também que

estes registos de estado não podem assim ser acedidos em modo *burst*, pois o bit B do *header* byte deve ser igual a zero, para este *header* byte ser interpretado pelo *transceiver* como um comando *strobe*. Caso contrário, este será interpretado como uma simples leitura do registo quando o bit $R/\overline{W}$ é igual a um. Se este bit for igual zero neste último caso, nada será feito uma vez que estes registos nesta gama de endereços não podem ser escritos.

Todo o espaço de endereçamento com os respectivos registos e comandos *strobe* podem ser consultados no anexo A.

5.1.2 Pacote de dados

O pacote de dados transmitido na comunicação sem fios tem o seguinte formato como se pode ver na figura 5.4:

- *Preamble*;
- Palavra de sincronização;
- 1 Byte de tamanho opcional;
- 1 Byte de endereço opcional;
- Campo de dados;
- 2 Bytes de informação da ligação opcionais (RSSI, LQI e CRC);

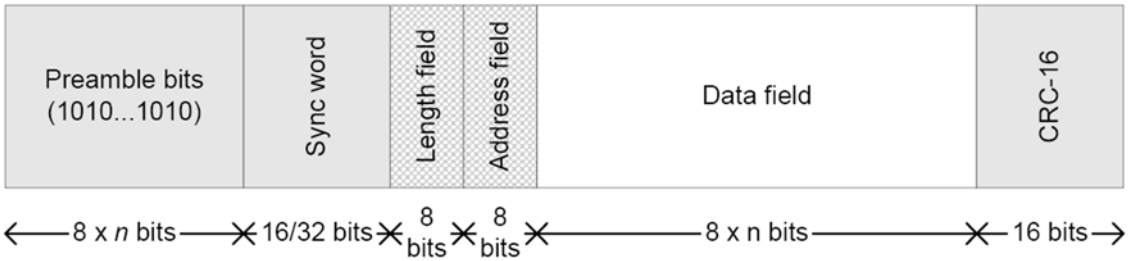


Figura 5.4: Formato dos pacotes.

O *preamble* consiste numa sequência alternada de zeros e uns (10101010...), que é transmitida sempre que é activado o modo de transmissão. O tamanho desta sequência pode ser programado. Quando o número de bytes do *preamble* é transmitido, se houver informação na *fifo* de transmissão é enviada a palavra de sincronização e depois a informação que contida na *fifo*. Se esta estiver vazia o *preamble* continuará a ser

transmitido até a *fifo* de transmisão ser escrita enquanto o modo de transmissão estiver activo.

O *preamble*, a palavra de sincronização são colocados na transmissão, processados e retirados na recepção automaticamente pelo *transceiver*. Os campos opcionais de tamanho e de endereço e os dois bytes finais com informação da ligação (RSSI, LQI e CRC) também são processados pelo *transceiver*, mas neste caso não são removidos da *fifo* de recepção.

No campo de dados é onde pode ser colocada a informação útil que se prentende enviar remotamente de um sistema para outro. O tamanho deste campo pode também ser programado, através de um tamanho fixo ou variável. Para este último caso, é utilizado um byte no início do campo de dados que define o tamanho do pacote.

Figura 5.5: Estrutura do campo de dados.

Nesta implementação é utilizado um tamanho fixo de 6 bytes de informação e deste modo o byte opcional de tamanho não é utlizado. Deste modo, o primeiro byte do campo de dados é o byte de endereço que indica qual o *transceiver* destino do pacote. Depois, como dentro do mesmo sistema implementado numa FPGA pode haver a necessidade de diferentes estruturas comunicarem remotamente com outro sistema, é utilizado um byte como identificador, de forma a haver um fácil reencaminhamento dos sinais dentro da FPGA. Os restantes 4 bytes (32 bits) são utilizados para enviar informação útil.

Foi escolhido este valor médio pois para a maioria das aplicações de controlo 32 bits chegam para a comunicação remota. Pois no caso em que são poucos bits necessários de transmitir, um tamanho grande do pacote conduzia a um grande desperdício na comunicação sem fios. Caso seja preciso enviar mais que esta informação, é possível dividi-la por diferentes identificadores ou também é possível alterar facilmente a implementação para o campo de dados ter capacidade até 63 bytes.

5.1.3 Configurador do *transceiver* CC1101

5.1.3.1 Gerador do relógio SPI - SCLK

Como já vimos anteriormente uma interface SPI pode ter quatro modos de operação, consoante a fase e a polaridade do relógio que sincroniza e gere a comunicação. Deste

modo, a fase e a polaridade do relógio gerado pela FPGA (*master*) tem de ser compatível com o relógio que o *transceiver* CC1101 foi programado para funcionar, para o correcto funcionamento da comunicação entre eles.

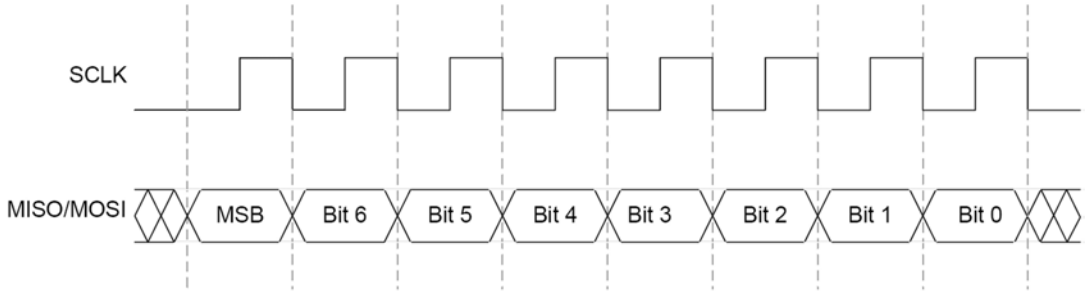


Figura 5.6: Fase e polaridade do relógio SPI[22].

Na figura 5.6 é possível observar a fase e polaridade do relógio SPI com que o *transceiver* CC1101 funciona. Deste modo, a fase do relógio deve ser gerado pela FPGA de maneira que a informação, ou seja, o bit esteja centrado no tempo na fase ascendente do relógio (quando o relógio passa de 0 para 1) e que a alteração para o próximo bit seja feita na fase descendente do relógio (quando o relógio passa de 1 para 0). A polaridade do SCLK deve ser programada de modo a que este fique a zero quando não há comunicação entre *master* e o *slave*.

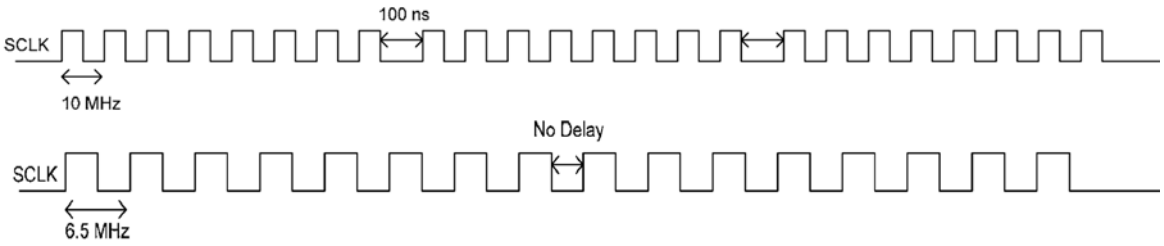


Figura 5.7: Frequência do SCLK.

O SCLK pode apresentar no máximo uma frequência de funcionamento de 10 MHz, no entanto é necessário adicionar neste caso atrasos de 100 ns entre cada byte transferido. Para não ser preciso adicionar qualquer atraso entre cada byte transferido a frequência de funcionamento máxima deverá ser 6.5 MHz. Como a velocidade de transferência de dados sem fios, será na ordem de dezenas de kbits por segundo, a frequência de 6.5 MHz é suficiente para este caso, simplificando assim a implementação do SCLK.

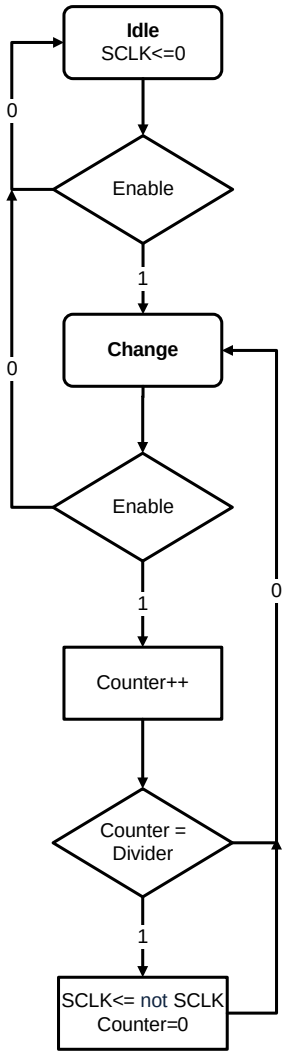


Figura 5.8: Diagrama de fluxo do gerador SCLK.

Esta implementação foi feita através de uma simples máquina de estados finitos (FSM) com apenas dois estados: o estado *Idle* e o estado *Change*. A máquina encontra-se no primeiro estado quando não há comunicação activa entre o *master* e o *slave*, indicado pelo sinal de entrada *Enable*. Aqui devido á polaridade referida atrás o SCLK fica sempre a zero até o sinal *Enable* seja activo pela camada acima para o início de uma comunicação, mudando assim para o estado *Change* onde é gerado um relógio enquanto o sinal *Enable* esteja activo. Para isto é utilizado um contador para controlar o tempo que demora a alteração do nível lógico do SCLK. A frequência deste SCLK pode ser alterada através de o parâmetro genérico de entrada *Divider*, onde a escolha deste valor é preciso ter em atenção a máxima frequência permitida de 6.5 MHz. O valor escolhido por defeito é 9, que para uma frequência do relógio 50 MHz respeita esta regra.

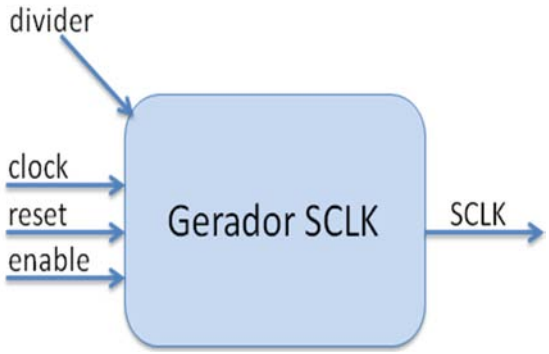


Figura 5.9: Interface do gerador SCLK.

5.1.3.2 Controlador da transferência de 1 byte

Nesta camada é feito o controlo e a sincronização dos restantes sinais (MOSI/MISO/CSn) da comunicação SPI, bem como a activação do SCLK no módulo anterior, na transferência de um byte entre o *master* e o *slave*.

Neste módulo também é utilizada uma máquina de estados finitos (FSM) para controlar a transferência de um simples byte. Por defeito, quando não há nenhuma comunicação activa a máquina estará no estado *Idle* até o sinal de entrada *transfer_cs* seja activo pela camada superior, passando assim para o estado *Init*, onde é activo o sinal *Chip Select* da interface SPI, de modo a seleccionar o *slave*. Em seguida vai para o estado *Wait_miso*, onde espera que o sinal MISO seja igual a zero (CHIP_RDYn), indicando que o *transceiver* está pronto para comunicar [22]. O sinal *flag* é para evitar este tempo de espera e consequente bloqueio, quando existe bytes transferidos consecutivamente, situação onde o sinal MISO não fica obrigatoriamente a zero no primeiro bit. Quando passa esta condição, é activado o SCLK e colocado na linha MOSI o primeiro bit a ser transferido, passando assim para o estado *Transfer*. Aqui é esperado até que o SCLK passe de zero para um ("*rising_edge*"), e quando isto acontece é processada a leitura da linha MISO, recebendo assim o bit que *slave* transmitiu. Depois transita para o estado *Change* onde espera que o SCLK passe de um para zero para efectuar a alteração para o próximo bit a ser transferido pela linha MOSI. Quando isto acontece volta novamente para o estado *Transfer* para a transferência do próximo bit. Deste modo, a máquina vai transitar entre estes dois estados sucessivamente até transferir o último bit, passando assim para o estado *Finish* onde espera o último ciclo do SCLK e assim pelo fim da última transferência. Por fim, quando chega ao estado *Load*, é desactivado o SCLK, bem como é sinalizado á camada superior que acabou a transferência de um byte através do sinal *Done_tick*.

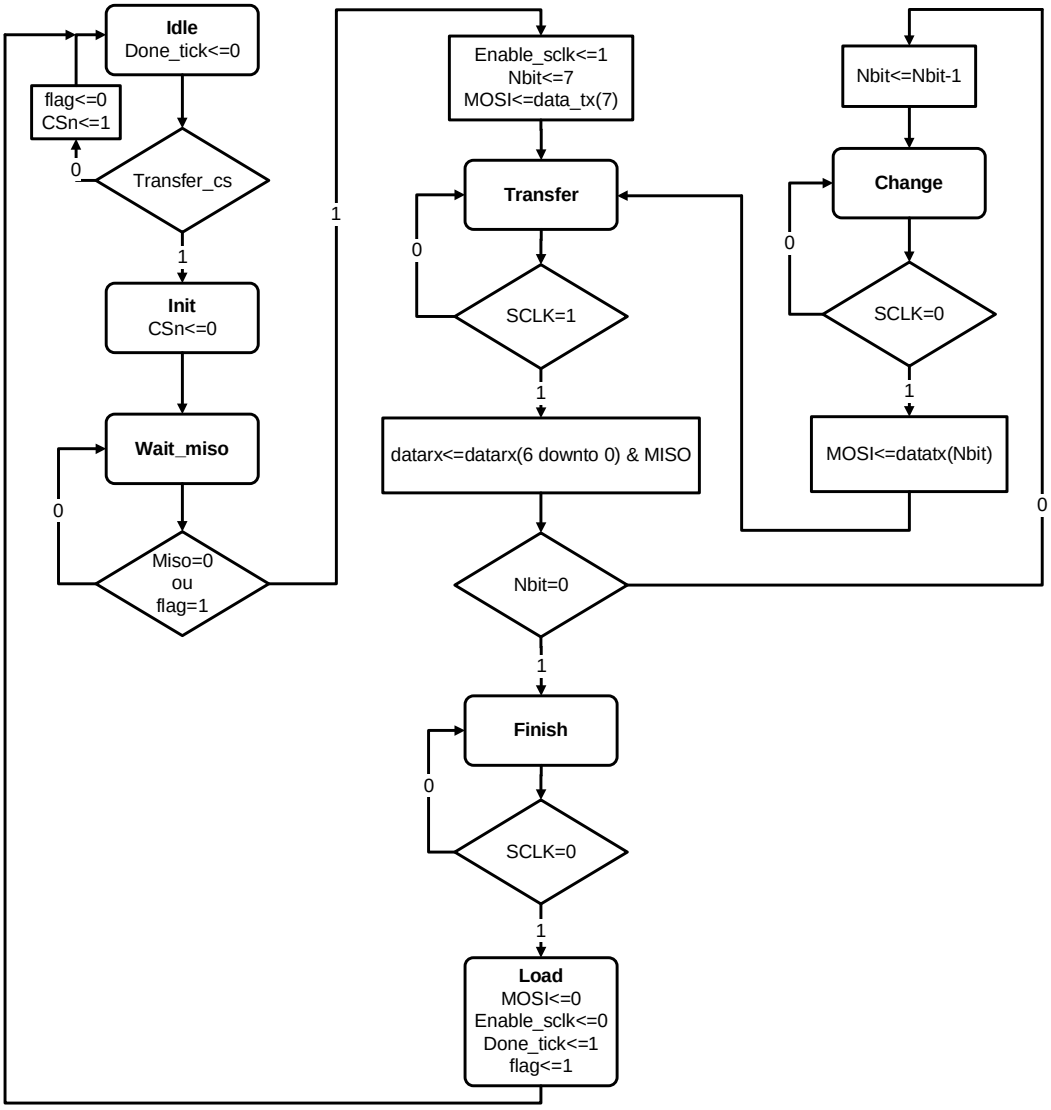


Figura 5.10: Diagrama de fluxo do controlador da transferência de um byte.

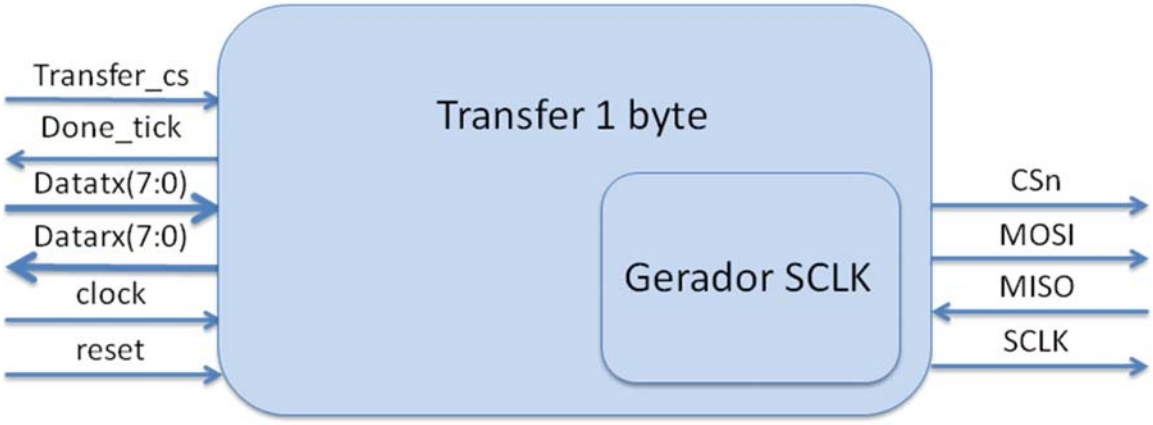


Figura 5.11: Interface do controlador da transferência de um byte.

5.1.3.3 Controlador de uma transferência SPI

Como já foi dito atrás, existem vários tipos de acessos que a interface SPI do *transceiver* aceita, como a leitura ou escrita, simples ou em *burst* e comandos *strobe*. Neste módulo é implementado os acessos simples de escrita ou leitura de um registo arbitrário do *transceiver*. Este tipo de acessos inclui a transferência de dois bytes entre o *master* e *slave*, utilizando para isso o módulo descrito anteriormente para transferir cada byte. Aqui também foi implementado o acesso através de comandos *strobe*.

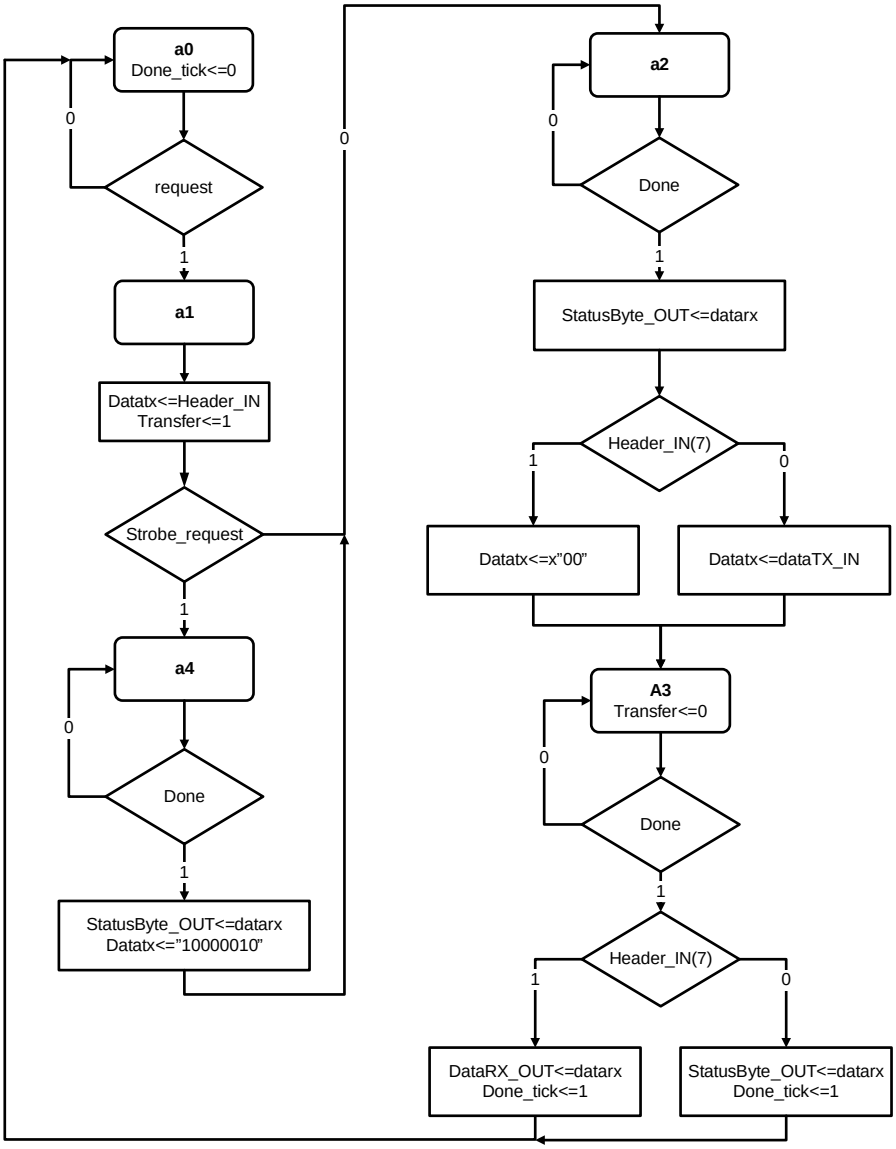


Figura 5.12: Diagrama de fluxo do controlador de uma transferência SPI.

Neste módulo é utilizada mais uma vez uma máquina de estados finitos (FSM) para controlar os diferentes acessos a interface SPI. Quando não há nenhum acesso

a interface SPI, a máquina estará no estado *a0* até que a camada de cima comece uma transferência, activando o sinal de entrada request, passando para o estado *a1*. Neste estado é enviado o *header* byte respectivo e é activada a transferência na camada inferior, que se responsabiliza pela transferência deste byte. Aqui consoante seja ou não um comando strobe passará para diferentes estados.

Quando é uma leitura ou escrita de um registo arbitrário do *transceiver*, a máquina passa para o estado a2, onde espera que transferência do *header* byte seja concluída. Depois disto, se o acesso for para escrever num registo, é enviado o byte respectivo para camada inferior, caso contrário se for uma leitura é enviado um byte nulo, pois este não vai ser usado pelo *transceiver*. Assim passa para o estado a3, onde ficará a espera da transferência do byte com a informação seja concluída. Quando isto acontece, se o acesso em questão for uma leitura, os dados recebidos são enviados para a camada superior. Em ambos os casos é sinalizada a camada superior que a transferência SPI acabou pelo sinal *Done_tick*.

Quando a máquina chega ao estado a1 e verifica que o sinal de entrada *Strobe_request* está activo, esta passa para o estado a4, onde espera que transferência do único byte seja concluída. Em seguida é feita uma leitura de um registo arbitrário, passando assim directamente para o estado a2, acabando como uma leitura normal já descrita anteriormente. O acesso em *burst* pode facilmente aqui ser implementado, neste módulo, bastando para isso adicionar alguns estados adicionais para esse efeito.

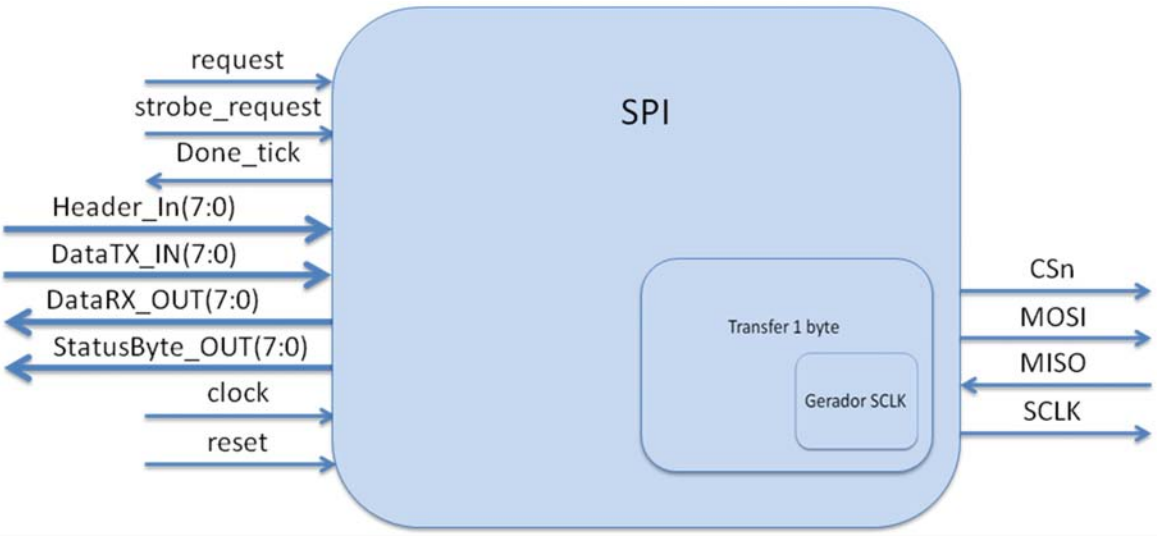


Figura 5.13: Interface do controlador de uma transferência SPI.

5.1.3.4 Controlo da comunicação

Nesta camada é feita a distinção automaticamente entre os acessos á interface SPI para leituras e escritas nos registos com os comandos *strobe* do *transceiver*.

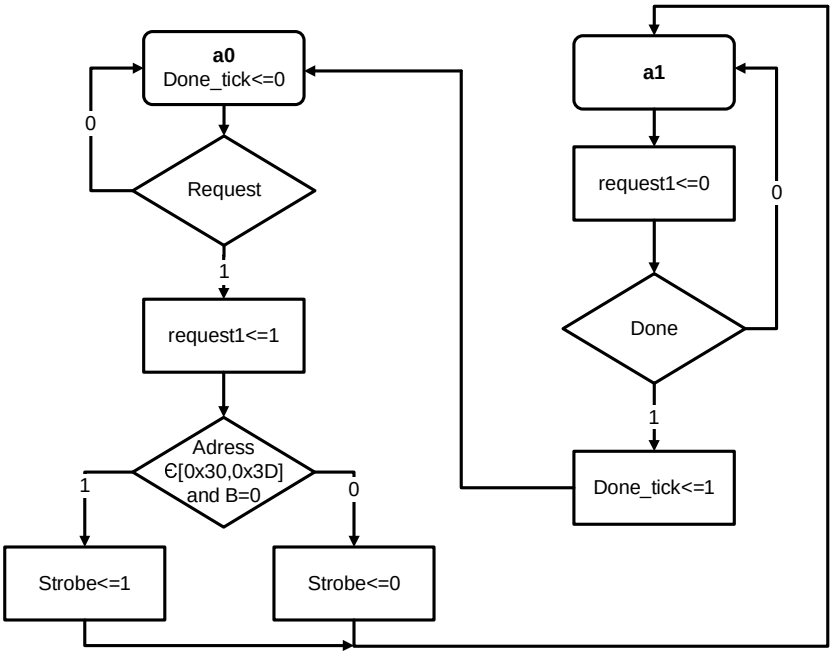


Figura 5.14: Diagrama de fluxo do controlador de comunicação.

Este módulo foi utilizado para simplificar o uso da interface SPI. Mais uma vez foi utilizada uma simples máquina de estados finitos (FSM) com apenas dois estados. Por defeito, quando não há comunicação a máquina fica sempre no estado *a0* até ser iniciada uma comunicação pela camada superior a esta através do sinal de entrada *Request*. Quando isto acontece, é activado o sinal *request* da camada mais abaixo, que se responsabiliza por todo o acesso SPI consoante o *header* byte enviado. Neste estado é também feita a verificação se é um comando *strobe* ou não, e em caso afirmativo (*Adress Є[0x30,0x3D]* e *B=0*) também é activada a entrada *strobe_request* da camada inferior, descrita anteriormente. Depois disto a máquina vai para o estado *a1*, onde vai apenas esperar que a comunicação seja concluída pela camada inferior. A semelhança do que acontece nas outras camadas, esta também sinaliza a camada superior quando a comunicação acaba pelo sinal *Done_tick*.

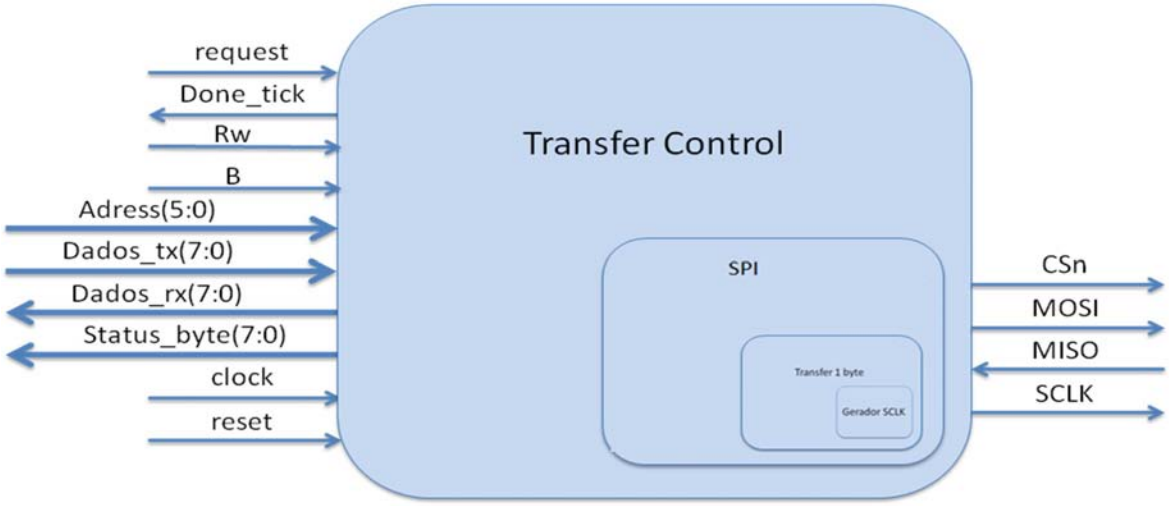


Figura 5.15: Interface do controlador de comunicação.

Com este módulo, é possível ter a toda abstracção em relação a todo acesso SPI, bem como o tipo de acesso (*Strobe*/leitura/escrita) como era pretendido. Assim, apenas e necessário especificar os campos do *header* byte (Rw-B-Adress), activar o *request* e esperar pelo fim da transferência (*done_tick*). No caso da leitura e escrita num registo também tem de ser usado os sinais *Dados_rx (7:0)* e *Dados_tx (7:0),* respectivamente.

5.1.3.5 Configurador

Neste módulo é feita toda a configuração inicial do *transceiver* CC1101. Com a ajuda do módulo anterior é feito o reset ao *transceiver*, e em seguida a configuração de todos os seus registos, bem como configurada a potência de saída (PATABLE), e por fim a comunicação SPI é colocada a disposição do controlador.

Aqui também foi utilizada uma máquina de estados finitos (FSM) para a implementação deste módulo. Como se pode observar na figura 5.16, no primeiro estado *reset_c1101*, é enviado o comando *strobe* SRES para fazer reset ao *transceiver*, de modo a este ficar no seu estado por defeito e com os seus registos por defeito. Depois disto é feita a configuração de todos os registos do *transceiver* desde o endereço 0 até o endereço 0x2E nos estados a1 e a2.

Para isto é utilizada uma memória ROM que contém todos os valores pretendidos para os respectivos registos do *transceiver*, já previamente calculados através do programa SmartRF Studio para um determinado modo de operação do *transceiver*. Os endereços desta memória ROM são exactamente iguais aos endereços dos registos do *transceiver*, permitindo assim o sinal *Adress* ser utilizado para endereçar ambos.

Quando acaba de configurar o último registo (0x2E), a máquina passará para o

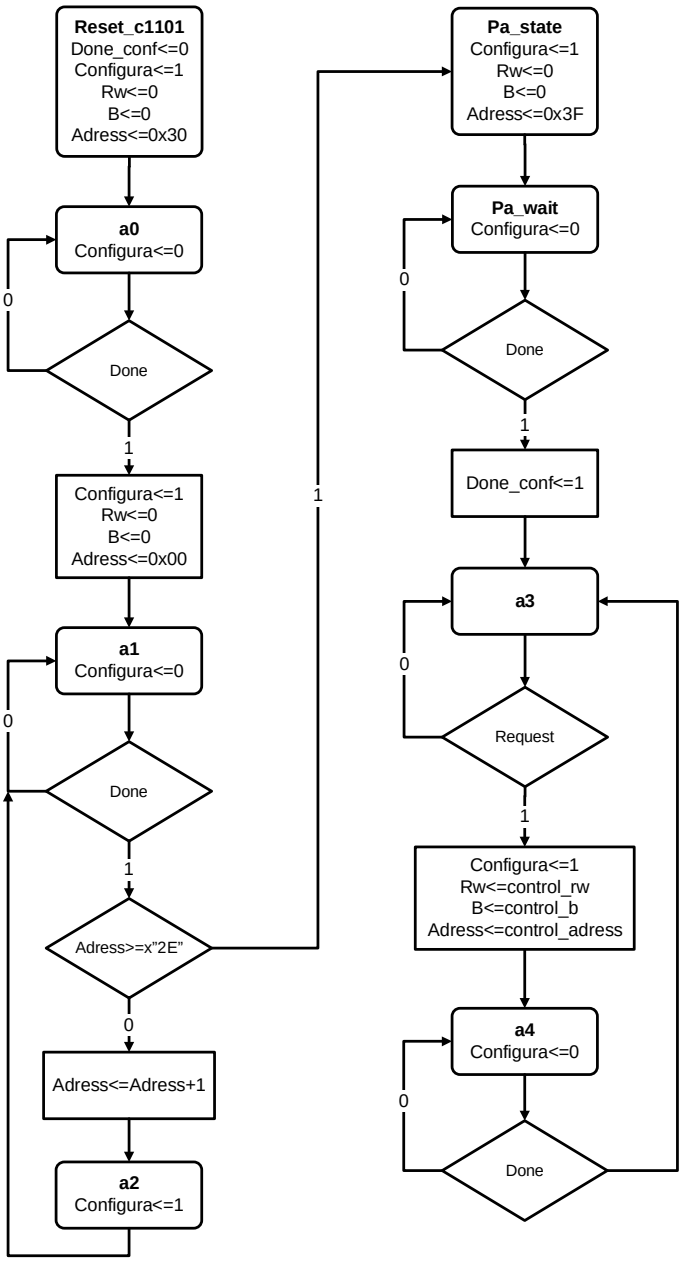


Figura 5.16: Diagrama de fluxo do configurador do *transceiver* CC1101.

estado *Pa_ state* onde configura a PATABLE com um valor pretendido para uma determinada potência de saída do *transceiver*. Assim quando esta última configuração acaba no estado *Pa_ wait*, é sinalizado para o resto do sistema que a configuração do *transceiver* está concluída através do sinal de saída *Done_ conf*, podendo a partir desse momento o *transceiver* ser utilizado pelo sistema para a comunicação sem fios propriamente dita, através de um controlador para esse efeito.

Deste modo, quando a configuração acaba, a máquina vai transitar sempre entre os estados a3 e a4, sempre que haja uma transferência por ordem do sistema, sendo que esta transferência será toda controlada por um controlador externo a este módulo.

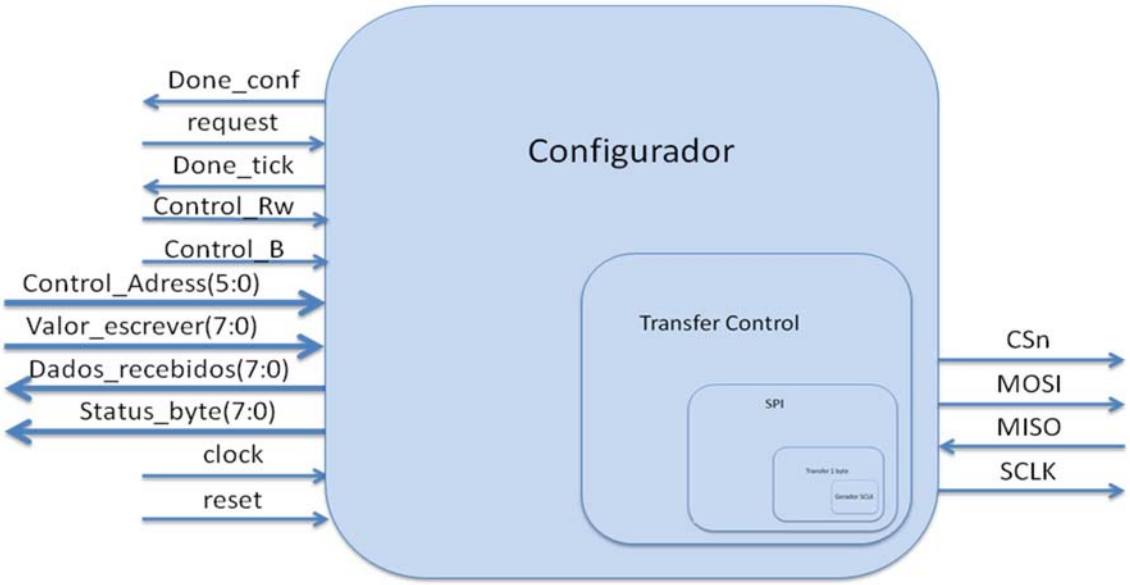


Figura 5.17: Interface do do configurador do *transceiver* CC1101.

5.1.4 Controlador do *transceiver* CC1101

Este módulo terá um papel fundamental na gestão da comunicação sem fios. Este comunica directamente com o configurador, deixando este programar o *transceiver* numa configuração específica, passando depois o controlo para este módulo. Aqui é feita a gestão do tempo em que o *transceiver* está no modo de recepção e transmissão, bem como a gestão das respectivas *fifos*.

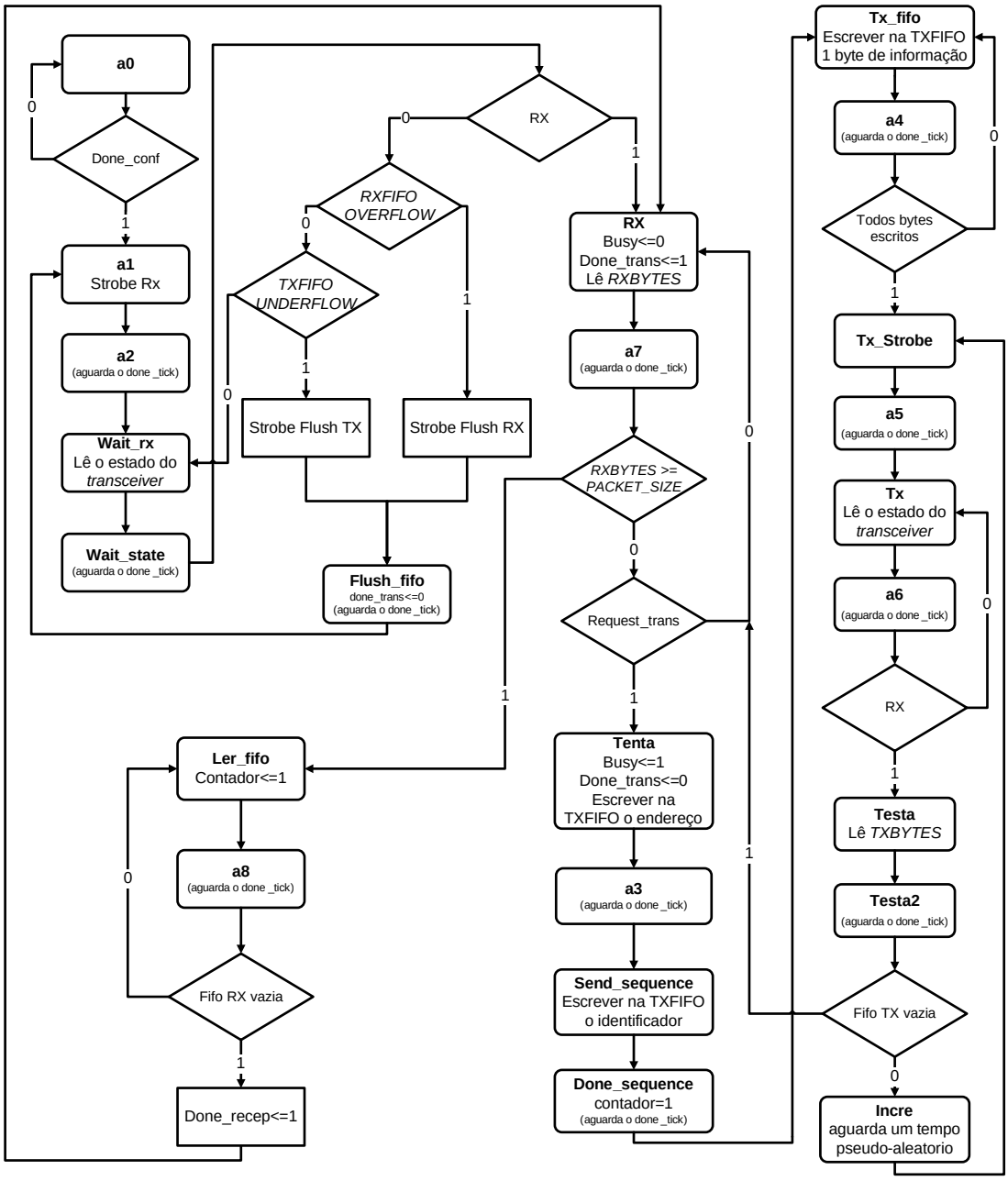


Figura 5.18: Diagrama de fluxo simplificado do controlador do *transceiver* CC1101.

Na figura anterior e possível observar o diagrama de fluxo simplificado do controlador do *transceiver* cc1101. Para a sua implementação também foi usada uma máquina de estados finitos (FSM). A principal característica deste controlador, centra-se no facto de colocar o *transceiver* sempre no modo de recepção (RX) por defeito, passando só para o modo de transmissão quando o sistema precisa de transmitir e se no momento o meio estiver livre (CCA). Caso esteja a receber no momento em que precisa de transmitir ou o meio esteja ocupado por outra transmissão, ele só transmitirá

depois.

A máquina começa no estado *a0* onde espera que a configuração do *transceiver* seja concluída passando assim para o estado *a1*. Aqui é enviado um comando *strobe* RX para o *transceiver* passar do estado *idle* para o estado de recepção. Em *a2* é apenas esperado que a este comando *strobe* seja transferido. Como o tempo transição entre os estados internos do *transceiver* é variável [20], é lido o registo que contém o estado interno do *transceiver* no estado *Wait_RX* até o *transceiver* estar efectivamente no estado de recepção. Se entretanto for detectado alguma fifo cheia, é enviado um *strobe* para limpar a *fifo* respectiva, voltando depois disto para o estado *a1*.

Deste modo, quando a máquina chega ao estado *Rx*, o *transceiver* também se encontra no estado de recepção, como era pretendido. Neste estado é lido constantemente o registo do *transceiver* que contém o número de bytes da *fifo* de recepção.

Quando este número de bytes atinge o número de bytes de um pacote, que no caso desta implementação é 8 (1 endereço + 5 campo de dados+ 2 informação da ligação), a máquina passa para o estado *Ler_fifo* onde vai ler e retirar sucessivamente todos os bytes contidos na *fifo* de recepção até esta ficar vazia. Os bytes que contêm informação útil (do 2^{o} ao 6^{o}) são enviados para o sistema. Quando isto acontece o resto do sistema é sinalizado que um novo pacote está disponível através do sinal *Done_recep* e a máquina volta para o estado *Rx*.

No estado *Rx/a7* quando não existe nenhum pacote completo na fifo de recepção, é verificado se existe algum pedido por parte do sistema para transmitir através do sinal de entrada *request_trans*. Em caso afirmativo a máquina passa para o estado *Tenta* onde é sinalizado o sistema pelo sinal *Busy* que o controlador já está a preparar o envio de um pacote. Assim é primeiro escrito na *fifo* de transmissão um byte com endereço a que se destina o pacote, depois é escrito o segundo byte que contém identificador. Depois disto, são escritos todos os bytes de informação, um de cada vez. Deste modo, quando todo o pacote já foi enviado para a *fifo* de transmissão do transceiver, a máquina atinge o estado *Tx_strobe*. Tal como o nome indica, aqui é enviado um comando *strobe* para o *transceiver* mudar para o estado de transmissão de modo a enviar o conteúdo da fifo de transmissão.

No entanto, este está programado para passar para o estado de transmissão só se o canal RF estiver livre ("*TX-if-CCA*") [20], pois se o canal estiver ocupado, será necessário enviar outro comando *strobe* para transmitir. Depois disto, a máquina passa para os estados *Tx/a6* onde lê continuamente o registo que contém o estado do *transceiver* e verifica se este está no estado de recepção. Convém referir, que o *transceiver* está configurado para quando acabar de transmitir o pacote voltar ao estado de recepção ("*TXOFF_MODE[1:0]*"). Em caso afirmativo, a máquina passa para

os estados *Testa/Testa2* onde verifica o número de bytes na *fifo* de transmissão. Se esta *fifo* não estiver vazia é porque o canal RF não estava livre quando foi enviada ordem de transmitir, e o *transceiver* não chegou a sair assim do estado de recepção. Neste caso, a máquina passa para o estado *Incre*, onde é esperado um tempo pseudo-aleatório de maneira que seja mais provável que o canal fique novamente livre. Depois deste tempo de espera a máquina volta o estado *Tx_strobe* onde repete os passos descritos até conseguir enviar todo o conteúdo da fifo de transmissão.

Quando isto acontece a máquina volta ao estado *Rx* e os sinais *Busy* e *Done_trans* são actualizados de forma a informar o resto do sistema que o pacote foi transmitido. Quando aparece no diagrama um *strobe* ou uma leitura ou escrita num registo, é activado o request do configurador e os sinais de entrada deste que vão formar o *header* byte respectivo são devidamente ligados. Estes foram omitidos para simplificação do diagrama de fluxo. Convém salientar aqui, que este controlador está desenhado em função de alguns registos de configuração (como por exemplo TXOFF_MODE[1:0], RXOFF_MODE[1:0]). Assim, para alteração destes registos também poderá ser necessário a alteração deste controlador.

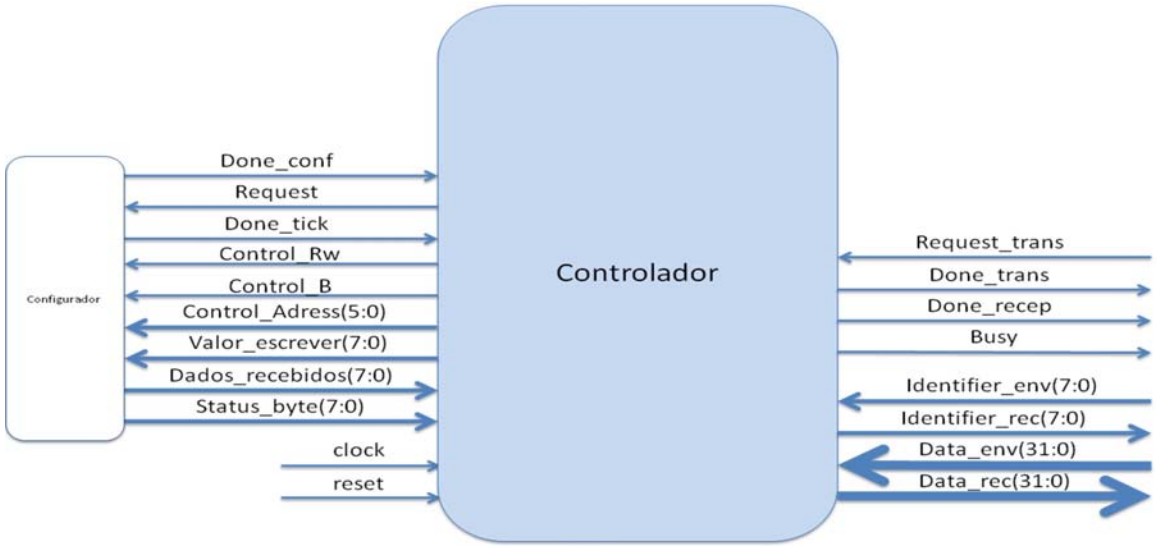


Figura 5.19: Interface do controlador do *transceiver* CC1101.

5.1.5 Controlo da transmissão

Este módulo vai interagir directamente com o controlador descrito anteriormente. A sua principal função é simplificar a interface com o controlador e gerir os pedidos de transmissão por parte do sistema.

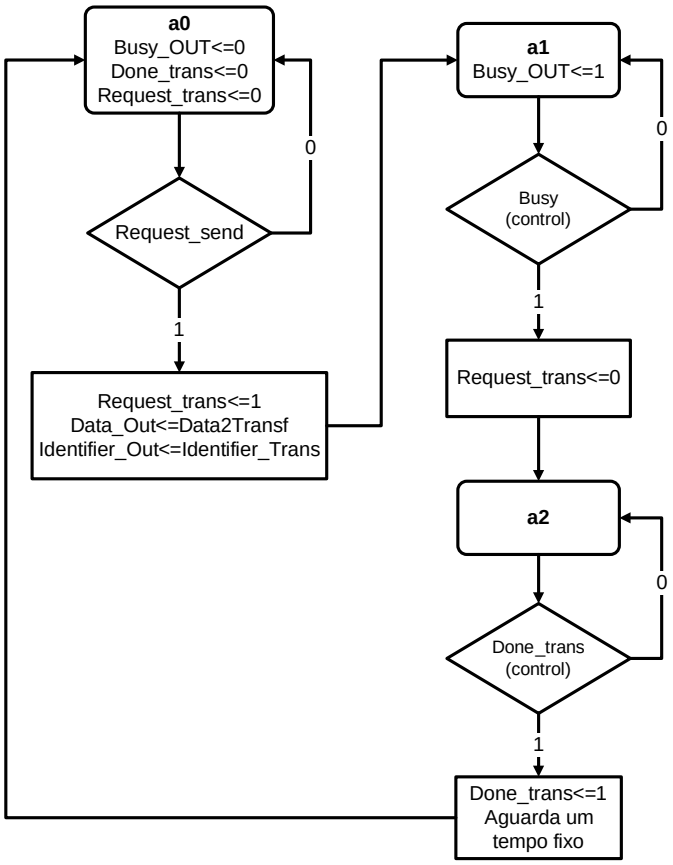


Figura 5.20: Diagrama de fluxo do controlo da transmissão.

Este módulo também foi implementado através de máquina de estados finitos (FSM). Começa no estado *a0*, onde está por defeito quando não há nenhuma transmissão de dados. Quando o sistema faz um pedido para enviar um pacote através do sinal de entrada *Request_ send*, é activado o sinal *Request_ trans* do controlador e é enviado para este o pacote de dados a transferir e o seu identificador, passando a máquina para o estado *a1*. Aqui é aguardado que o controlador receba o pedido e inicie o envio do pacote (quando o controlador passa estado *tenta*). Assim quando o controlador já está a processar o envio do pacote (*Busy*=1), o sinal *Request_ trans* é desactivado e a máquina passa para o estado *a2* onde espera que o envio seja concluído, ou seja, quando o sinal *Done_ trans* do controlador volte a um.

Depois do envio, é aguardado um tempo fixo antes de voltar ao estado *a0* para estar disponível para transmitir um novo pacote. Este tempo de espera foi inserido, pois como é uma comunicação bidireccional, outros *transceivers* também podem ter a necessidade de transmitir informação, dando assim a possibilidade de outro transmitir, ao libertar o canal RF por algum tempo.

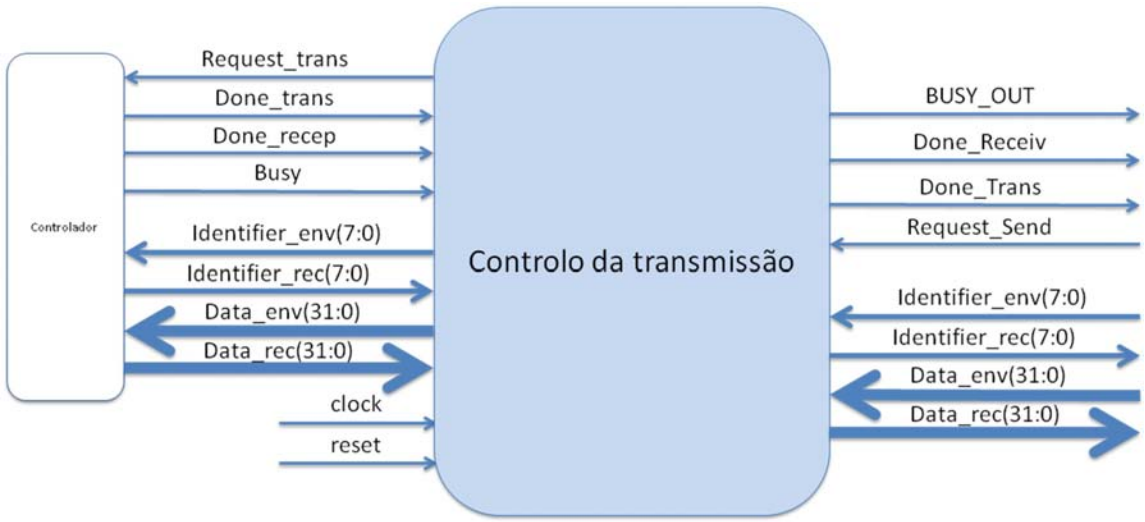


Figura 5.21: Interface do controlo da transmissão.

5.1.6 Protocolo

Como se trata de uma comunicação bidireccional, pode acontecer que ambos tentem transmitir ao mesmo tempo, e deste modo pode ocorrer a perda de pacotes. Para evitar esta situação, foi implementado um protocolo que utiliza *Acknowledge* de modo a dar mais fiabilidade a esta comunicação sem fios.

Deste modo, quando é transmitido um pacote, o sistema fica a espera da recepção de um *Acknowledge* do *transceiver* que recebeu o pacote, de forma a ter a certeza que o pacote chegou ao destino. Se este não receber o *Acknowledge* num determinado intervalo de tempo, o mesmo pacote é enviado outra vez. Assim quando é recebido um pacote é necessário enviar um *Acknowledge* para o *transceiver* origem saber que o pacote chegou ao destino e não reenviar o mesmo pacote outra vez.

A implementação deste protocolo obviamente tira eficiência á comunicação sem fios, na medida em que só metade dos pacotes enviados é informação útil, sendo que o resto são *Acknowledge*, no entanto torna a ligação sem fios muito mais confiável.

Na figura 5.22 é demonstrado um exemplo do funcionamento deste protocolo. Inicialmente só *transceiver* A tem pacotes para enviar. Este só envia o pacote 2 depois de ter recebido o *Acknowledge* do pacote 1. Entretanto o *transceiver* B tem o pacote 3 para transmitir, no entanto este não vai transmitir, pois o canal RF já esta ocupado pelo transceiver A, como já foi explicado anteriormente na parte do controlador. Deste modo, o *transceiver* B só vai conseguir transmitir depois do *transceiver* A acabar de transmitir o pacote 2 e desocupar assim o canal RF. Este envia primeiro o pacote 3, antes do *Acknowledge* do pacote 2 pois, este sistema já se já se encontra nos estados

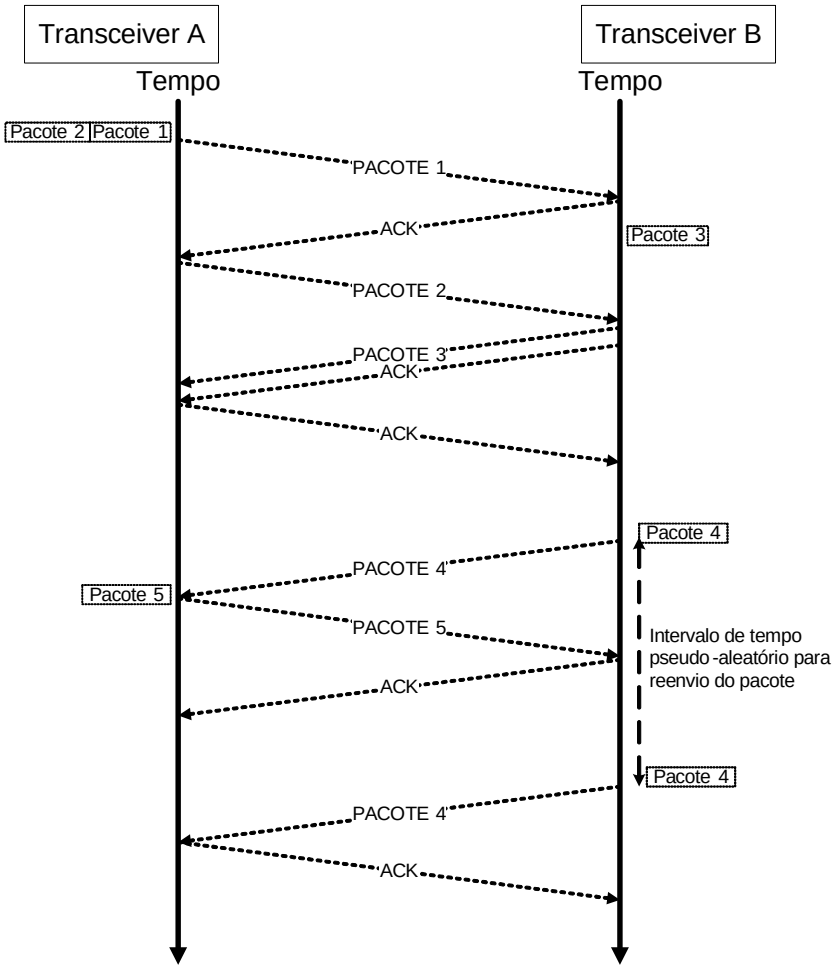


Figura 5.22: Exemplo do funcionamento do protocolo com *acknowledge*.

onde controlador tenta transmitir, ou seja, o pacote 3 está na fifo de transmissão do *transceiver* B. Assim depois do envio deste pacote é aguardado um tempo de espera imposto pelo módulo anterior (controlo da comunicação) antes de ser possível transmitir de novo.

Se o tempo de transmissão do pacote 3 demorar mais tempo que o tempo de espera introduzido na camada anterior, o *Acknowledge* do pacote 2 é enviado primeiro que o *Acknowledge* do pacote 3 (como o que acontece na figura 5.22). No entanto pode acontecer o contrário, ou seja, se o tempo de transmissão for menor que o tempo de espera introduzido, o *Acknowledge* do pacote 3 é enviado primeiro que o *Acknowledge* do pacote 2.

No caso da figura é possível observar que o *transceiver* A não recebe o pacote 4, pois está a transmitir o pacote 5. Como *transceiver* B não recebe assim o *Acknowledge* do pacote 4, este reenvia-lo outra vez depois de um intervalo de tempo pseudo-aleatorio.

Para a implementação deste protocolo, este pode ser dividido em duas partes: a transmissão e recepção.

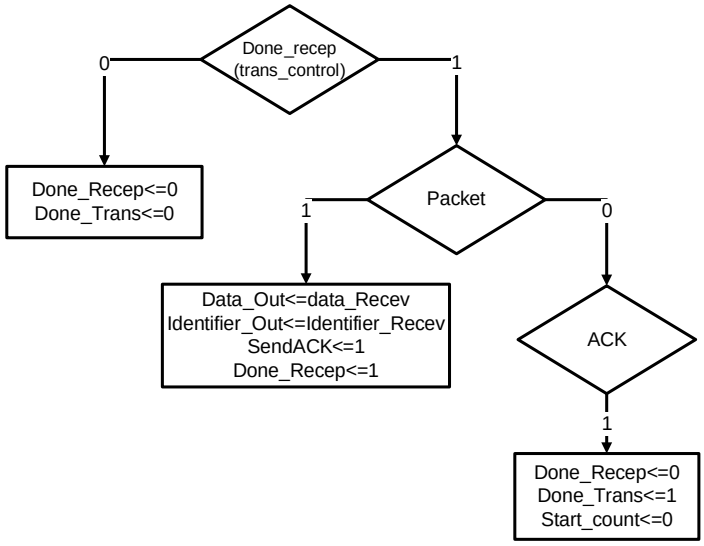


Figura 5.23: Diagrama de fluxo do protocolo na parte de recepção.

Na implementação do protocolo na parte de recepção é usado um simples processo, que verifica continuamente a recepção de novos pacotes através do sinal de entrada *Done_recep*, que informa quando um novo pacote e recebido e lido do *transceiver* pelos módulos anteriormente descritos. Assim quando este sinal avisa a recepção de um novo pacote, é verificado se este pacote é um *Acknowledge* ou é um pacote com informação útil. Se for informação útil os dados recebidos são reencaminhados para o sistema, e este é também avisado da recepção do novo pacote pelo sinal de saída *Done_recep*. Quando isto acontece também é activado o sinal *SendACK*, de modo a ser enviado o respectivo *Acknowledge* na próxima transmissão deste sistema. Quando é recebido um *Acknowledge* é sinalizado ao sistema, o fim da transmissão do pacote pelo sinal de saída *Done_trans* bem como é desactivada a contagem do tempo para o reenvio do pacote, pois neste caso não será preciso mais reenviar, uma vez que já foi recebido o *Acknowledge* esperado.

A parte de transmissão do protocolo que se pode observar na figura 5.24, foi implementada usando uma máquina de estados finitos (FSM). Inicialmente e por defeito quando não há transmissões a fazer a máquina encontra-se no estado *Idle*. Aqui é verificado continuamente quando o controlo da transmissão está disponível para transmitir através do seu sinal *Busy*, e em caso afirmativo, é verificado se tem de enviar um *Acknowledge* ou algum pacote de informação do sistema. Neste caso é dada prioridade ao *Acknowledge* que é activado no processo de recepção descrito anteriormente através do sinal *SendACK*.

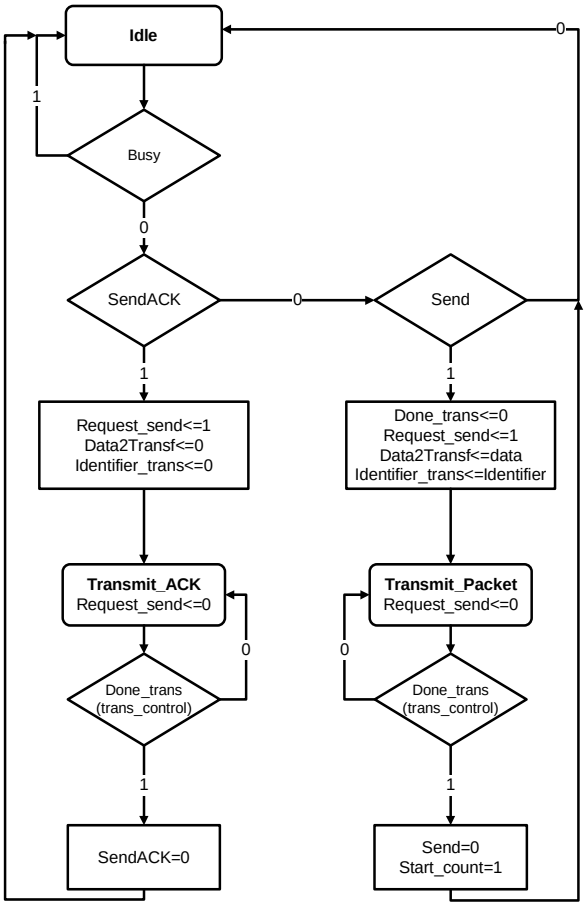


Figura 5.24: Diagrama de fluxo do protocolo na parte de transmissão.

Deste modo, é feito um pedido de transmissão ao módulo anterior a este (controlo da transmissão) através do sinal *Request_send* e a máquina passa assim para o estado *Transmit_ACK* onde espera que a transmissão do *Acknowledge* seja concluída pelos módulos anteriores. Assim que isto acontece, o sinal *SendACK* é desactivado e a máquina volta ao seu estado inicial *Idle*.

Quando o sistema está disponível para transmitir e não há nenhum *Acknowledge* que precisa ser enviado, se o sinal *Send* estiver activo, é feito um pedido de transmissão ao controlo da transmissão e o pacote de informação útil respectivo é enviado para os módulos anteriores. Deste modo a máquina passa para o estado *Transmit_Packet* onde espera também que a transmissão do pacote seja concluída. Depois disto o sinal *Send* é desactivado e um contador é activado através do sinal *Start_count*, voltando a máquina ao seu estado inicial *Idle*.

Este contador permite a contagem de um intervalo de tempo pseudo-aleatório para o reenvio do pacote caso entretanto o sistema não receba um *Acknowledge*. Deste modo, o sinal *Send* pode ser activo de duas formas: pelo sistema, quando precisa de

enviar um pacote com informação, ou pelo contador, quando este atinge um tempo de espera de um *Acknowledge* superior ao intervalo de tempo pré-estabelecido (pelo gerador aleatório).

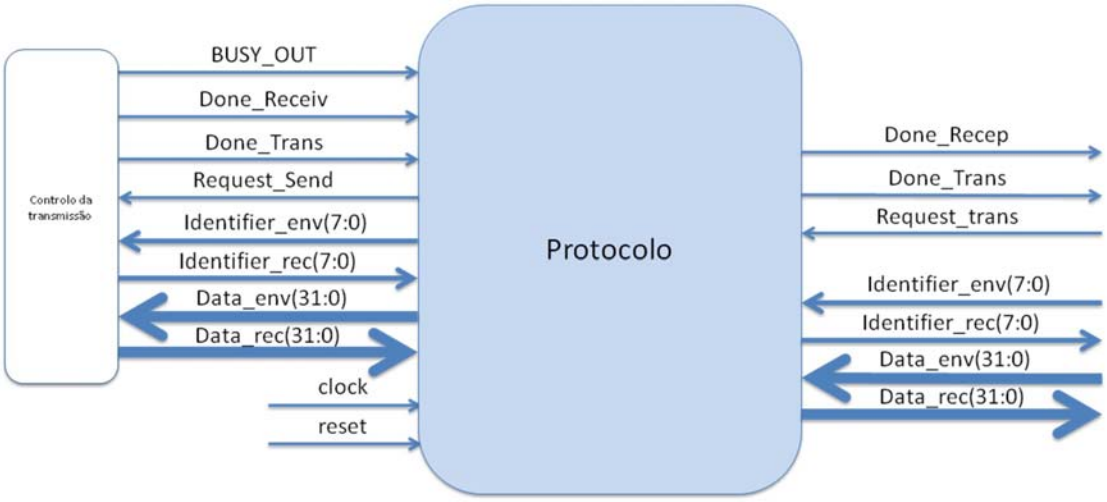


Figura 5.25: Interface do protocolo.

5.2 Módulo *Transceiver*

Neste módulo são ligados todos os blocos anteriormente descritos, de forma juntar todos estes módulos num só bloco mais simples e intuitivo de usar em qualquer sistema implementado em FPGAs, que necessite de uma interface que permita uma comunicação sem fios.

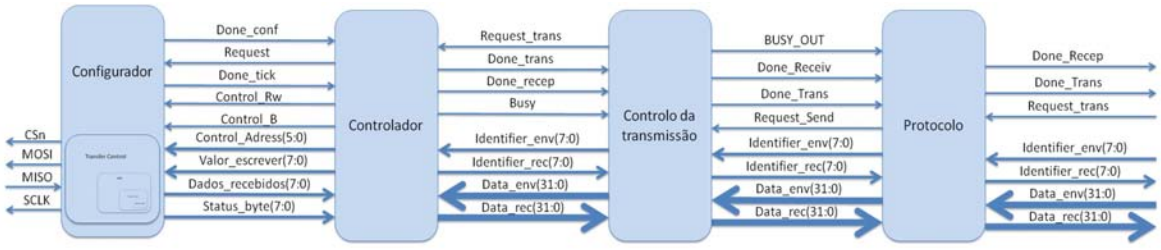


Figura 5.26: Ligações entre todos os módulos incluídos dentro do módulo *transceiver*.

Para a construção deste bloco, além dos módulos anteriores, também foi utilizada uma pequena máquina de estados finitos (FSM), que permite gerir mais facilmente os pedidos de transmissão por parte do sistema ao módulo protocolo.

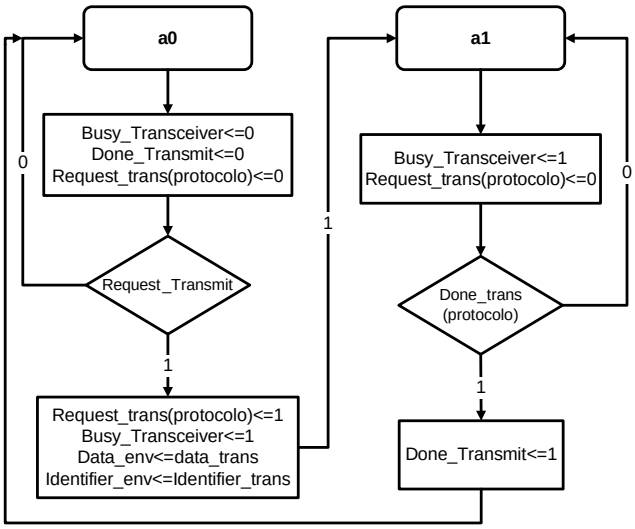


Figura 5.27: Diagrama de fluxo da transmissão no módulo *transceiver*.

Como se pode observar na figura 5.27, esta máquina utiliza apenas dois estados, *a0* e *a1*. O primeiro estado é o seu estado por defeito, para quando não existe nenhum pacote a ser transmitido actualmente, estando assim disponível para transmitir caso necessário ($Busy\_transceiver <= 0$). Aqui é verificado se existem pedidos de transmissão por parte do sistema através do sinal *Request_ transmit*, em caso afirmativo a máquina passa assim para o estado a2, onde é esperado que a transmissão do pacote seja concluída (com a recepção do *Acknowledge* respectivo). Neste estado o sistema sabe que o *transceiver* está ocupado ($Busy\_transceiver <= 1$) e assim não aceita novos pedidos de transmissão. Quando a transmissão é concluída, o sistema também é avisado e a máquina volta ao seu estado inicial, ficando assim disponível outra vez para transmitir.

Na parte de recepção, não foi necessário fazer mais nenhum tipo de controlo, e assim as saídas do bloco do protocolo são ligadas directamente á saída do bloco do *transceiver*. Estas saídas são apenas o sinal *Done_ Receiv* que avisa quando um pacote de informação foi recebido e pode ser lido pelo sistema e o próprio pacote de informação com o identificador respectivo, *Data_ Receiv* e *Identifier_ receiv*.

5.3 Módulo CC1101

A principal limitação desta ligação é obviamente a sua taxa de transferência. Para evitar a perda de pacotes foram introduzidas duas *fifos*, de modo a guardar os pacotes que chegam a uma velocidade superior á taxa de transmissão desta ligação sem fios (relativamente baixa).

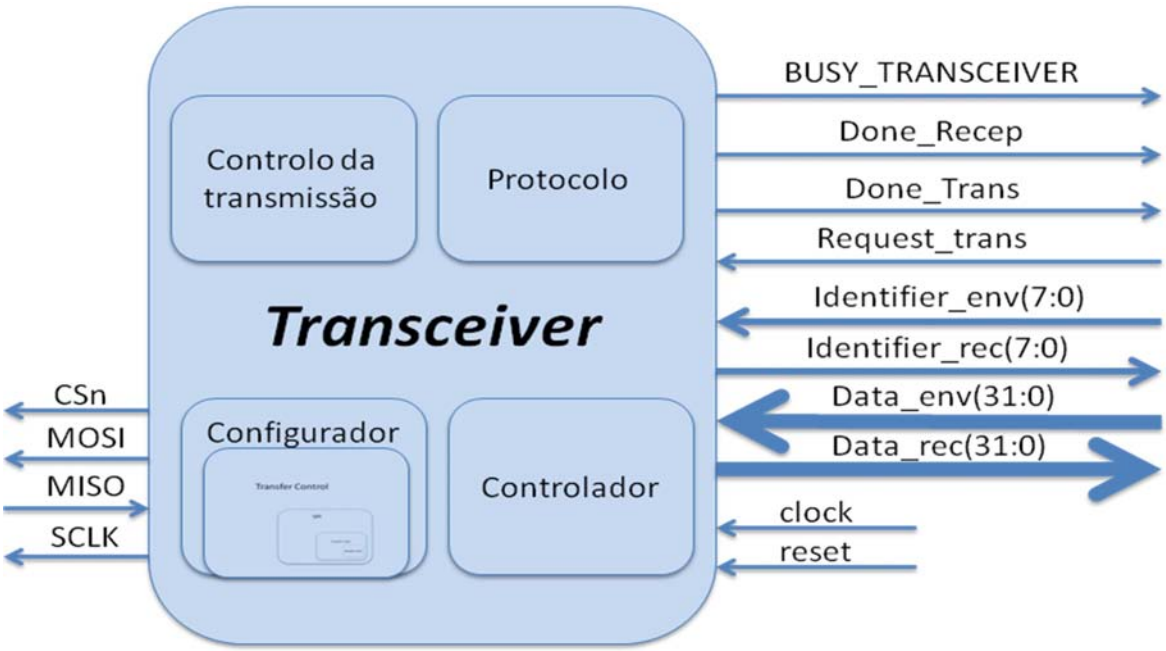


Figura 5.28: Interface do módulo *transceiver*.

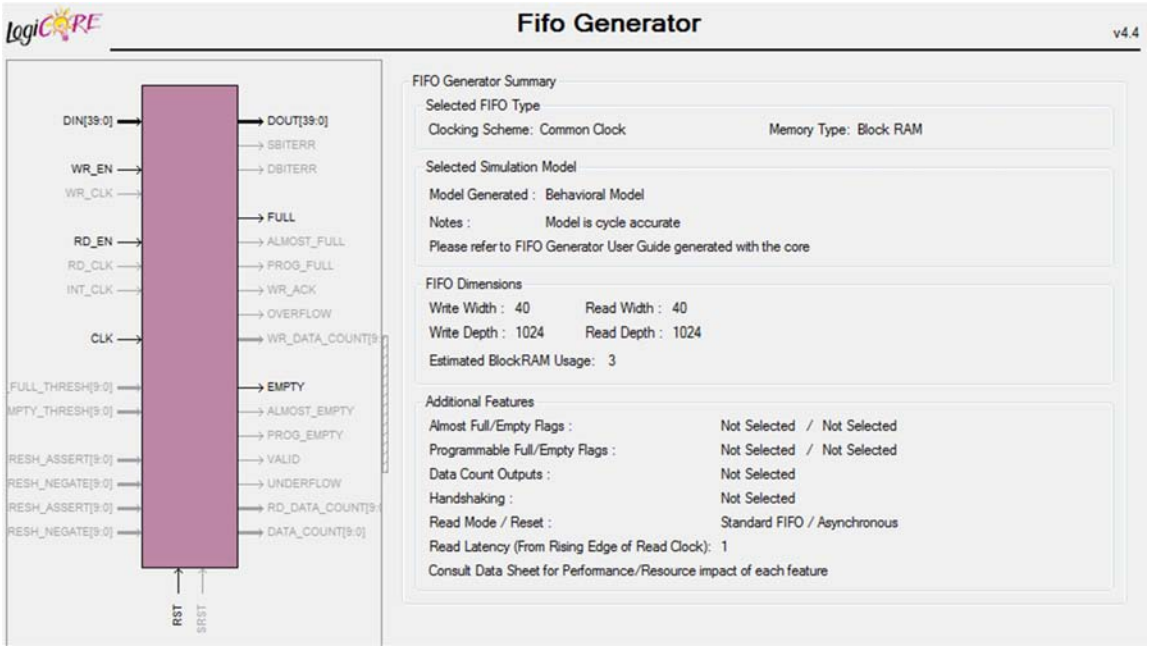


Figura 5.29: Interface e resumo das características das *fifos* utilizadas.

Deste modo, a introdução destas duas *fifos* oferece á comunicação sem fios criada, maior robustez e adaptabilidade a um maior número de sistemas possível. A introdução de uma *fifo* de transmissão é assim essencial, pois permite guardar e colocar em fila de espera os pacotes que chegam para ser enviados, quando o *transceiver* já se en-

contra ocupado. Uma *fifo* de recepção também pode ser bastante importante pois não obriga uma parte do sistema está sempre a verificar se recebeu informação, lendo a *fifo* apenas quando precisa ou antes de esta encher, no entanto, em muitos sistemas esta *fifo* não precisa de ser usada, pois toda informação recebida pode ser reencaminhada directamente.

Para implementação destas *fifos* foi utilizado um núcleo de propriedade intelectual, o *Fifo Generator v4.4* (Xilinx LogiCORE™ IP) [51].

Na figura 5.29 é possível observar, as opções seleccionadas para *fifos* utilizadas neste módulo. Estas são implementadas em *Block RAM*, evitando assim o uso excessivo de recursos lógicos da FPGA. Cada palavra guardada nesta *fifo* tem um tamanho de 40 bits, de modo a guardar o pacote de informação com 32 bits e o seu identificador respectivo de 8 bits. Cada *fifo* tem capacidade de 1024 palavras de 40 bits, conferindo assim uma boa capacidade de armazenamento, nos momentos mais críticos com elevado tráfego de pacotes. No entanto, este valor pode facilmente ser alterado consoante a aplicação. Convém referir também que a leitura da *fifo* apresenta uma latência de um ciclo de relógio, que é necessário ter em atenção, para uma correcta leitura desta.

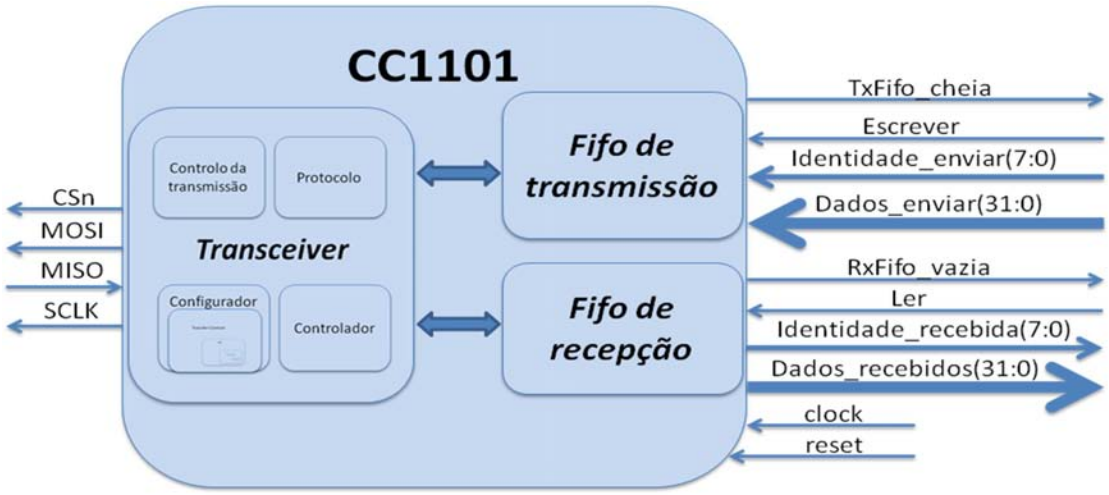


Figura 5.30: Interface final do módulo CC1101.

Deste modo, quando o sistema precisa de transmitir, em vez de comunicar directamente com o módulo *transceiver*, apenas escreve os pacotes que precisa de enviar na *fifo* de transmissão, e assim, é esta fifo é que vai interagir directamente com o módulo *transceiver*. Num processo, é verificado continuamente o sinal *BUSY_ TRANSCEIVER*, e quando este estiver a zero, ou seja disponível para transmitir, a *fifo* transmissão se não estiver vazia é lida, e no próximo ciclo do relógio o valor da saída desta *fifo* é colocado á entrada do módulo *transceiver* para enviar, através dos sinais *Data_ env* e

Identifier_ env.

O mesmo acontece na recepção de pacotes, sempre que um pacote de informação é recebido, este é escrito na *fifo* de recepção, sendo que o sistema depois lê da *fifo* de imediato ou não, consoante este esteja configurado.

Assim com a introdução destas *fifos* de recepção e transmissão é fornecido ao sistema que utiliza esta interface sem fios uma maior abstracção, pois este só vai ter de interagir com as respectivas *fifos*. A organização hierárquica do projecto ISE do módulo CC1101 final que fornece uma interface sem fios pode ser vista na figura 5.31.

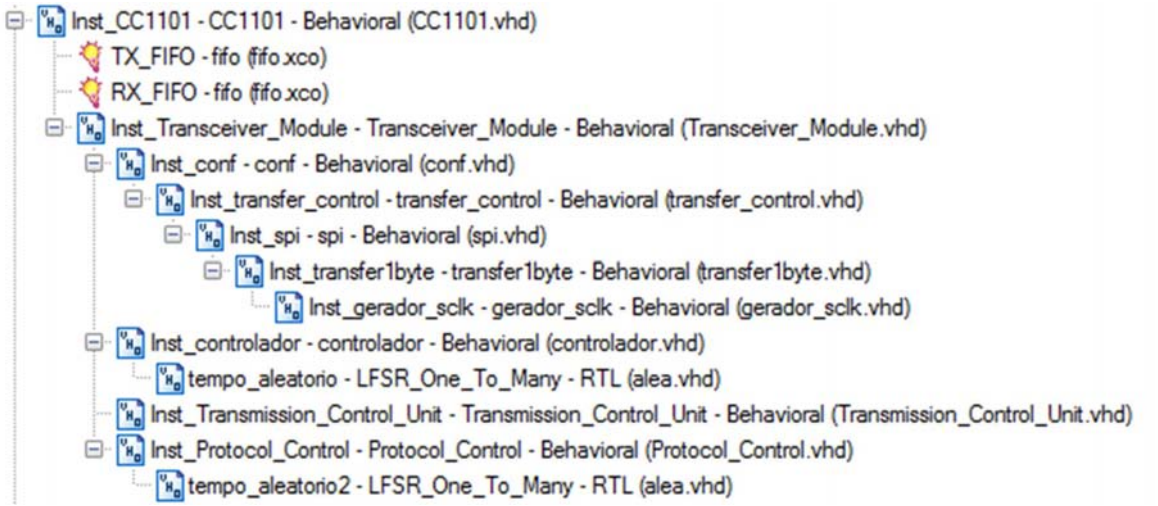


Figura 5.31: Organização hierárquica do projecto no ISE.

Convém referir que para a correcta comunicação entre todos estes módulos anteriores foi necessário ter em atenção os ciclos dos relógios dos processos correspondentes de forma a haver uma intercalação das fases ascendestes (*rising_ edge*) e descendentes (*falling_ edge*) do relógio, entre os módulos que estão ligados directamente.

5.4 Resultados

Todos estes módulos foram sintetizados e implementados nas FPGAs das duas placas disponíveis para este trabalho, Nexys2 e Celoxica RC10. No entanto, os resultados podem ser melhorados, através de algumas optimizações do código (como por exemplo armazenar os registos de configuração em *Block* RAMs).

Módulo	**LUTs**	Nexys2	RC10	***Slices***	Nexys2	RC10
Gerador do SCLK	43	0.5%	0.2%	40	0.9%	0.3%
Transferência de 1 byte	38	0.4%	0.1%	32	0.7%	0.2%
Transferência SPI	38	0.4%	0.1%	27	0.6%	0.2%
Controlo da comunicação	7	0.08%	0.03%	5	0.1%	0.04%
Configurador	83	0.9%	0.3%	53	1.1%	0.4%
Controlador	254	2.7%	1.0%	182	3.9%	1.4%
Controlo da Transmissão	53	0.6%	0.2%	85	1.8%	0.6%
Protocolo	200	2.1%	0.7%	161	3.5%	1.2%
Módulo *Transceiver*	4	0.04%	0.02%	45	1.0%	0.3%
Módulo CC1101	110	1.2%	0.4%	170	3.7%	1.3%
Total	**830**	**8.9%**	**3.1%**	**800**	**17.1%**	**6.0%**

Tabela 5.1: Utilização de cada módulo da interface sem fios.

Na tabela 5.1, é possível observar os recursos lógicos necessários para a implementação desta interface, bem como a percentagem da lógica utilizada em duas FPGAs de baixo custo e consequentemente, de baixa capacidade. Para a implementação das *fifos* no módulo CC1101 com o tamanho descrito anteriormente também foi necessário utilizar 3 *Blocks* RAMs para cada *fifo*. No entanto, este valor pode variar consoante o tamanho seleccionado para cada *fifo*.

Estes resultados mostram que esta interface pode assim ser utilizada numa basta gama de aplicações, pois esta não é muito dispendiosa em termos de recursos físicos.

Os primeiros módulos, cujo objectivo principal era uma correcta comunicação com o *transceiver* através da interface SPI, foram testados através da análise dos sinais da interface SPI, com a ajuda de um analisador de lógica. Para isso foi construído, um circuito de *debug* próprio para a medição destes sinais (como se pode observar no Anexo B), para evitar o risco de danificar o *transceiver*. Os sinais observados aqui, permitem concluir que existe uma correcta comunicação entre a FPGA e o *transceiver*, como era esperado e descrito anteriormente.

Os módulos das camadas superiores á medida que foram desenvolvidos, foram testados através do envio de pacotes com valores de contadores nos dois sentidos, de forma a poder avaliar a ligação sem fios nos dois sentidos. Aqui foi possível observar se há bloqueio na ligação em algum sentido, bem como se há perda de pacotes através da comparação dos contadores.

Para isto, as *fifos* de transmissão do módulo CC1101 são constantemente escritas de forma a medir a máxima taxa de transferência possível. Para estes testes, o *transceiver* foi configurado com as seguintes parâmetros principais:

- Modulação: GFSK;

- *Datarate*: 100 KBaud;
- Tamanho do pacote de dados fixo: 6 Bytes;
- *Autoflush* CRC;
- Verificação de endereço;
- 4 Bytes de palavra de sincronização;
- 6 Bytes de *preamble*.

Teste 1 1 metro / 3600 seg	FEC desactivo		FEC activo	
	Transceiver A	*Transceiver* B	*Transceiver* A	*Transceiver* B
Pacotes enviados	520,049	520,079	318,337	318,363
Último valor enviado	520,048	520,078	318,336	318,362
Pacotes recebidos	520,079	520,049	318,363	318,337
Último valor recebido	520,078	520,048	318,362	318,336
Acknowledges enviados	520,079	520,049	318,363	318,337
Reenvios	0	0	0	0
Taxa de transmissão	5.8 kbit/s	5.8 kbit/s	3.5 kbit/s	3.5 kbit/s
Taxa (incluindo acesso ao meio)	74 kbit/s		45 kbit/s	
Taxa (incluindo *Acknowledges*)	23.1 kbit/s		14.1 kbit/s	
Taxa de transferência	11.6 kbit/s		7.1 kbit/s	

Tabela 5.2: Teste da comunicação sem fios a uma distância de 1 metro.

Os resultados obtidos no teste 1 referem-se a condições óptimas, pois os dois *transceivers* encontram-se perto um do outro (1 metro de distância), em linha de vista, havendo transmissão nos dois sentidos. Desta forma, estes resultados mostram que todos os pacotes enviados nos dois sentidos foram recebidos no destino. Aqui também é possível observar que todos os *Acknowledges* foram recebidos, uma vez que não ocorreu nenhum reenvio.

Assim, a taxa transferência máxima obtida para esta implementação no total dos dois sentidos é de 11.6 kbit/segundo. Esta taxa refere-se apenas á informação contida no campo de dados, que é útil ao sistema baseado em FPGA, ou seja, foram apenas considerados os 32 bits de cada pacote e os 8 bits com o identificador respectivo. Como era esperado, pode-se verificar que com os *Acknowledges* enviados esta taxa de transferência duplicaria.

Se forem considerados todos os bits enviados na transmissão de um pacote, isto é, os bytes do *preamble*, da palavra de sincronização e o byte de endereço, obtemos uma taxa de 74 kbit/segundo, o que já se aproxima do *datarate* teórico de 100 KBaud.

O resto da diferença entre estes dois valores deve-se sobretudo ao atraso imposto no módulo de controlo da transmissão, de modo a ser possível os dois sistemas transmitirem alternadamente e por cada pacote de dados ter apenas 6 bytes, quando as *fifos* do *transceiver* tem capacidade até 64 bytes.

Para o mesmo teste, mas com corrector de erros activo (FEC - *Forward Error Correction*) obteve-se uma taxa de transferência mais baixa, como era esperado, pois com esta opção activa, os pacotes tem de passar por mais fases de processamento, o que prolonga o tempo de transmissão. Estes resultados mostram que esta opção tem um impacto negativo de cerca de 40% na taxa de transferência.

Teste 2 240 metros / 900 seg	FEC desactivo		FEC activo	
	Transceiver A	*Transceiver* B	*Transceiver* A	*Transceiver* B
Pacotes enviados	118,973	115,787	79,839	79,927
Último valor enviado	118,972	115,786	79,838	79,926
Pacotes recebidos	120,681	124,731	79,951	79,854
Último valor recebido	115,786	118,972	79,926	79,838
Acknowledges enviados	120,681	124,731	79,951	79,854
Reenvios	11,666	11,765	64	40
Taxa de transmissão	5.3 kbit/s	5.2 kbit/s	3.5 kbit/s	3.6 kbit/s
Taxa (incluindo acesso ao meio)	68.3 kbit/s		45.5 kbit/s	
Taxa (incluindo *Acknowledges*)	21.3 kbit/s		14.2 kbit/s	
Taxa de transferência	10.5 kbit/s		7.1 kbit/s	

Tabela 5.3: Teste da comunicação sem fios a uma distância de 240 metros.

O segundo teste foi realizado para uma distância de 240 metros entre os *transceivers*. Nestas condições, já é possível verificar algumas falhas na transmissão de alguns pacotes. Assim é possível observar o funcionamento do protocolo implementado, pois neste caso são reenviados cerca de 10% dos pacotes. Aqui também pode-se observar o efeito do corrector de erros FEC, pois com esta opção activa, o número de reenvios é quase nulo.

No entanto, mesmo para esta distância não compensa utilizar esta opção, uma vez que a taxa transferência continua a ser bastante inferior, como se pode verificar na tabela 5.3.

Com estas definições, foi possível estabelecer uma comunicação estável até uma distância de sensivelmente 270 metros[52]. Contudo, nesta distância a ligação já depende muito da posição das antenas e da linha de vista (desobstrução do primeiro elipsóide de Fresnel[53]).

Em relação ao consumo de energia do *transceiver*, este varia desde 6.4 mA até 9.8 mA, dependendo da sua potência de saída que pode ser programada entre -30 e 10

dBm.

Deste modo, é possível estabelecer uma ligação sem fios entre FPGAs com os requisitos inicialmente pretendidos, ou seja, uma ligação bidireccional de baixo custo e consumo, confiável e com uma taxa de transferência e distância razoável.

5.5 Interacção remota no parque de estacionamento

Como o sistema do parque de estacionamento funciona como era esperado (dentro da mesma FPGA) e com a ajuda da interface RF criada, é possível agora implementar a comunicação sem fios entre a unidade de controlo central do parque com a unidade de controlo dos carros.

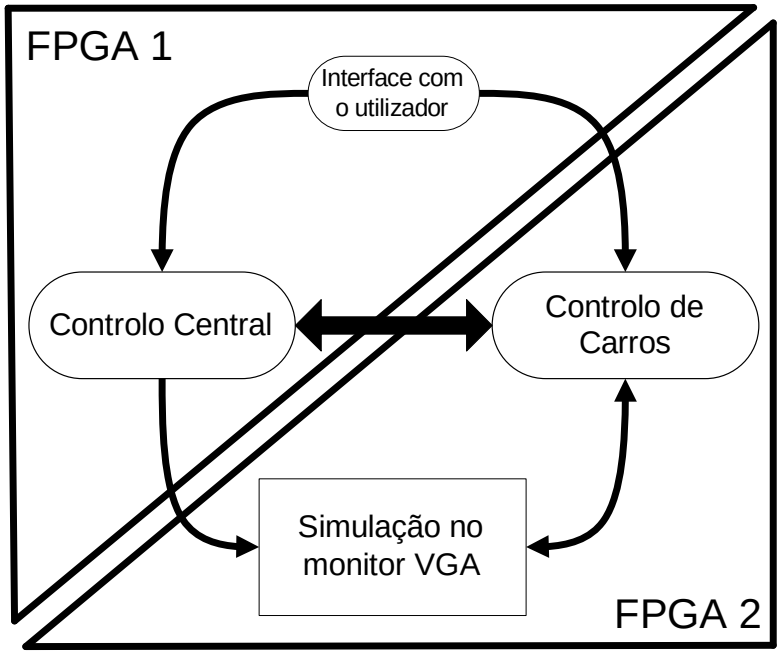


Figura 5.32: Divisão entre duas FPGAs do sistema do parque de estacionamento automático.

Deste modo, como a comunicação mais importante que se pretende testar é a interacção remota entre o controlo central com cada carro, os blocos do sistema foram divididos entre diferentes FPGAs como pode ser visto na figura 5.32. Assim numa FPGA é implementado controlo central e interface com o utilizador que permita controlar todo o sistema, enquanto que noutra FPGA é implementado o controlo de cada carro, bem como a simulação de todo o parque de estacionamento no monitor VGA.

Nos resultados da síntese de todo sistema do parque de estacionamento numa só FPGA, observou-se que os blocos do controlo dos carros e da simulação no monitor

VGA ocupam muito mais recursos lógicos que os blocos do controlo central e a da interface com o utilizador. Por esse motivo, para a implementação do controlo central e interface com o utilizador foi utilizada a placa de desenvolvimento Nexys 2 que contém uma FPGA com três vezes menos capacidade lógica que a FPGA da placa Celoxica RC10.

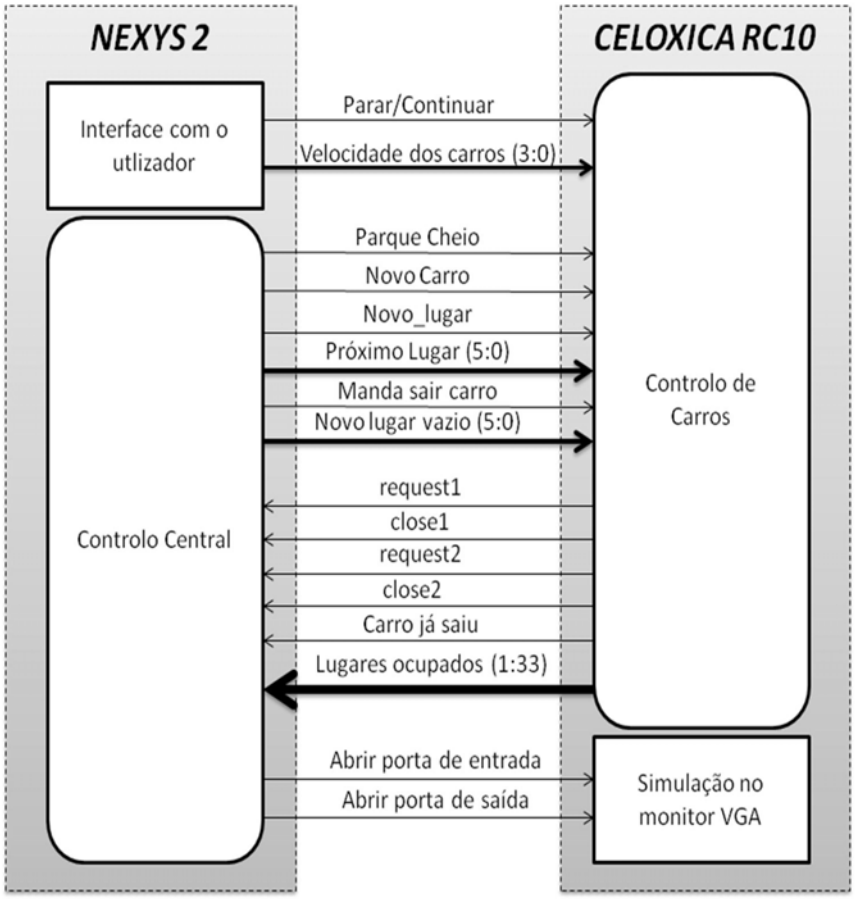


Figura 5.33: Sinais transmitidos sem fios.

Com esta divisão dos blocos do sistema entre as duas placas de desenvolvimento, os sinais que deste modo necessitam ser transmitidos remotamente são os que se podem observar na figura 5.33.

Convém referir que os sinais que representam números inteiros foram devidamente convertidos num número mínimo de bits necessário para representar toda a gama de valores que estes sinais podem ter. Assim os sinais de controlo que têm de ser enviados no sentido da Nexys 2 para a Celoxica RC10 são:

- Parar/continuar (1 bit);
- Velocidade dos carros (4 bits);

- Parque Cheio (1 bit) ;
- Novo Carro (1 bit) ;
- Novo Lugar (1 bit) ;
- Próximo lugar (6 bits) ;
- Mandar sair carro (1 bit) ;
- Novo lugar vazio (6 bits) ;
- Abrir porta de entrada (1 bit) ;
- Abrir porta de saída (1 bit) ;

No total é necessário enviar 23 bits neste sentido. Como na interface RF proposta permite enviar pacotes de dados de 32 bits (4 bytes), é possível enviar todos estes sinais de controlo referidos atrás num só pacote.

No sentido contrário, ou seja, no sentido da Celoxica RC10 para a Nexys 2, os sinais de controlo que precisam ser enviados remotamente são:

- *request1* (1 bit)
- *close1* (1 bit)
- *request2* (1 bit)
- *close2* (1 bit)
- Carro já saiu (1 bit)
- Lugares ocupados (33 bits)

Assim neste sentido é necessário enviar ao todo 38 bits. Como cada pacote contém no máximo 32 bits, é necessário dividir esta informação por dois pacotes, com identificadores diferentes de modo que no sistema de recepção a informação seja reencaminhada correctamente. Outra possibilidade seria aumentar o número de bytes de cada pacote de dados, nos módulos do *transceiver* para poder ser enviada toda a informação num pacote. No entanto esta solução alternativa é mais limitada, pois *transceiver* pode enviar no máximo 64 bytes em cada pacote e implicaria alterações apesar de simples nos módulos da parte da interacção remota.

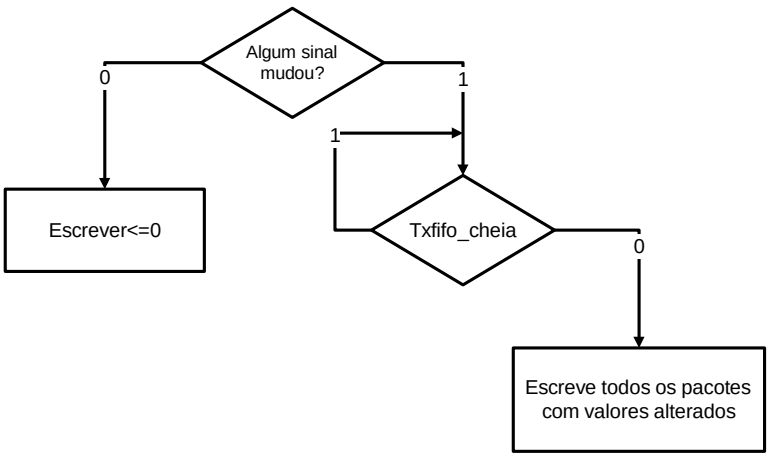


Figura 5.34: Diagrama de fluxo para a transmissão dos sinais.

Para o envio nos dois sentidos, o método utilizado para a transmissão dos sinais é o mesmo. Para isto, é utilizado um processo que verifica continuamente o estado de todos os sinais descritos anteriormente. Quando é detectada a alteração do valor de um sinal qualquer, o pacote onde está incluído este sinal é escrito na fifo de transmissão do módulo CC1101 caso esta *fifo* não esteja cheia. Quando são alterados vários sinais de diferentes pacotes, todos estes pacotes também são escritos na fifo de transmissão segundo uma dada ordem.

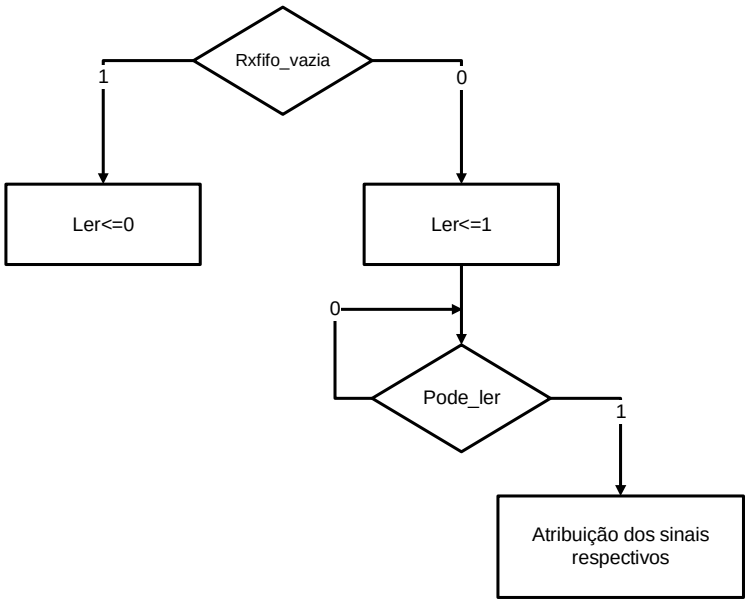


Figura 5.35: Diagrama de fluxo para a recepção dos sinais.

Na recepção dos sinais o método é semelhante á transmissão. Aqui também é utilizado um simples processo, que verifica continuamente o estado da *fifo* de recepção

do bloco CC1101 através do seu sinal de saída *RxFifo_vazia*. Assim quando esta *fifo* recebe algum pacote, esta é lida de imediato e os dados recebidos são reencaminhados e atribuídos aos sinais respectivos em função também do identificador recebido com cada pacote de dados.

Convém referir que estes sinais agora ligados remotamente através desta interface sem fios, apresentam atrasos adicionais, sendo preciso ter cuidado com alguns tipos de ligações. Assim os sinais desta ligação que antes funcionavam como impulsos (*request1*, *request2*, *close1*, *close2*, *já_saiu* e *novo_lugar*), tem de ser adaptados de forma a não depender destes atrasos adicionais. Para isto em vez de se considerar para estes sinais o seu valor, são consideradas apenas as suas transições (ascendentes), de maneira a haver uma comunicação correcta e síncrona entre os sistemas.

Capítulo 6

Conclusões e trabalho futuro

Sumário

Neste capítulo são apresentadas as conclusões principais desta dissertação fazendo uma análise global aos resultados e aos objectivos atingidos. No final são apresentadas algumas possibilidades para um eventual trabalho futuro.

6.1 Análise global

Na primeira parte do trabalho foi implementado um *buffer* de prioridade com uma estrutura de dados baseada numa árvore binária. Esta abordagem permitiu explorar o potencial de modelos avançados de FSM na implementação de algoritmos modulares, hierárquicos, recursivos e paralelos em *hardware*. Com isto, também foi possível verificar a ocupação e libertação de memória dinamicamente em *hardware*.

Em seguida, para a avaliação e teste do *buffer* de prioridade, bem como para a comparação com outras estruturas de dados e implementações diferentes, foi implementado um sistema que funciona como uma ferramenta para a verificação deste tipo de componentes reutilizáveis.

Aqui também foi construído um sistema embutido para controlo automático de um parque de estacionamento, onde é utilizado o *buffer* de prioridade para a gestão dos lugares de estacionamento, permitindo desta forma, ser demonstrada uma aplicação prática do bloco construído anteriormente.

Com o desenvolvimento deste sistema foi também possível verificar algumas capacidades importantes que as FPGAs oferecem, uma vez que através do mesmo dispositivo foi possível para além da implementação do próprio controlo, o desenvolvimento de um ambiente que permite uma simulação visual do sistema implementado e assim anal-

isar diferentes arquitecturas de controlo, evitando deste modo a necessidade de um ambiente físico naturalmente bem mais caro.

Numa segunda parte do trabalho foi desenvolvido um módulo reutilizável que fornece uma interface para comunicação remota (sem fios) entre circuitos implementados em FPGA. Esta interface sem fios foi construída com o auxílio de um *transceiver* RF de baixo custo, que através de uma ligação SPI que este fornece, é possível a sua programação e controlo. O *transceiver* escolhido revelou ser uma boa escolha, visto que com este foi possível assegurar uma ligação sem fios de baixo custo, e com uma distância e taxa de transferência pretendida e principalmente confiável, uma vez que não é perdida nenhuma informação útil.

Por fim, foi demonstrada a utilização deste módulo que fornece uma interacção remota no sistema de controlo automático do parque de estacionamento, para o estabelecimento de uma ligação sem fios entre as unidades de controlo dos carros com a unidade de controlo central do parque. Toda a simulação agora dividida entre duas FPGAs diferentes, funciona como era esperado, sempre que é possível uma ligação física RF. No entanto, esta simulação pode agora ser suspensa pelas limitações físicas da ligação sem fios, como por exemplo a distância, taxa de transferência, interferências, etc. Apesar disto, sempre que a ligação sem fios entre as duas FPGAs pode ser restabelecida, a simulação também é restabelecida normalmente sem perda de informação como era pretendido.

A utilidade das *fifos* incluídas na interface com o *transceiver,* também pode ser verificada em aplicações como no caso do parque de estacionamento, quando é aumentada a velocidade da simulação. Neste sistema, a comunicação entre os carros e o controlo central pode ser assim induzida para valores superiores à capacidade da ligação sem fios implementada, e deste modo verifica-se que nos momentos de maior tráfego vários pacotes a serem acumulados nas *fifos* de transmissão respectivas, sendo estas depois esvaziadas nos momentos seguintes de menor tráfego como era esperado.

Estas *fifos* em conjunto com o protocolo com *Acknowledge* implementado, demonstraram fornecer uma grande robustez a ligação sem fios, uma vez que, mesmo que a ligação sem fios falhe por motivos de limitações físicas, não são perdidos nenhuns pacotes com informação útil, sempre que esta ligação é restabelecida mais tarde. Para isto o tamanho das *fifos* deve ser dimensionado de modo a poderem guardar o máximo de pacotes que podem ser acumulados no caso de falha na ligação sem fios ou no caso de um tráfego de pacotes elevado.

Este sistema mostrou assim ser ideal para o teste dos principais blocos reutilizáveis desenvolvidos neste projecto, nomeadamente o *buffer* de prioridade e a interface que permite interacção remota, pois requer uma comunicação bidireccional bastante se-

gura e fiável, pois a perda de apenas um pacote específico pode ser fatal ao correcto funcionamento da simulação.

Convém referir também que o uso de blocos IP reutilizáveis como o *fifo generator* e o *block memory generator*, foram bastante úteis, pois permitiu um desenvolvimento mais rápido deste projecto.

Todos os circuitos anteriores foram implementados em *hardware*, verificados e testados, o que confirma que todas as funções e características desejadas estão correctas do ponto de vista lógico. As ferramentas visuais que foram desenvolvidas podem ser utilizadas para experiências no desenvolvimento de sistemas para análise de diferentes arquitecturas, num ambiente mais próximo da realidade e assim mais fiável.

Por fim, é possível concluir que os principais objectivos propostos neste projecto foram atingidos com sucesso, uma vez que foi possível demonstrar uma interacção remota em sistemas embutidos implementados em FPGA, através de uma ligação sem fios de baixo custo e confiável. Aqui também foi demonstrado o potencial da implementação em *hardware* de algoritmos de controlo modulares, hierárquicos, recursivos e paralelos e como as FPGAs podem ser utilizadas para simulação visual dos sistemas cada vez mais complexos implementados nelas próprias.

6.2 Trabalho futuro

Com o desenvolvimento deste trabalho, é possível destacar várias ideais de investigação futura:

- Investigação e optimização de um *buffer* de prioridade com outro tipo de arquitectura e estrutura de dados para comparação e avaliação de diferentes tipos abordagens. Um exemplo disso pode ser a implementação com uma estrutura de dados baseada numa lista biligada de prioridades.

- Estudo de estratégias alternativas para a gestão de prioridades, como por exemplo a comparação de soluções baseadas em processador com soluções baseadas em máquinas de estados finitos.

- Optimização da interacção remota, no sentido de incluir vários tipos de configurações RF, de modo a poder mudar as características da ligação sem fios em tempo real. Isto pode ser muito útil por questões de segurança e adaptabilidade ao meio. Para esse efeito, os valores dos registos de configuração do *transceiver* contidos num ROM devem ser passados para blocos de memória embutida (*blocks RAM*s) contidos nas FPGAs de modo a ocupar menos recursos e assim poder

armazenar vários conjuntos de valores correspondentes a ligações sem fios com diferentes características.

- Aplicação da interacção remota para aplicações onde são necessárias ligações sem fios mais específicas. Os módulos construídos até o configurador podem ser utilizados sem alteração do que está implementado e através destes pode ser construído um controlador mais específico e optimizado para uma determinada aplicação como por exemplo para uma ligação unidireccional, em que a sua implementação seria bastante mais simples. No entanto, para isto pode ser usado também o controlador já implementado como *template* para um desenvolvimento mais rápido do projecto.

- Estudo e comparação deste tipo de implementação com uma implementação baseada num microprocessador embutido numa FPGA para a comunicação com o *transceiver* RF.

- Testar esta ligação sem fios, para o caso de interacção remota entre vários sistemas com esta interface sem fios, uma vez que neste trabalho só foi testada na prática a comunicação sem fios entre dois sistemas. Para isto, pode ser utilizado o campo de endereço contido em cada pacote de dados, de modo, a uma filtragem correcta dos pacotes entre os diferentes sistemas remotos.

Apêndice A

Características internas do *transceiver* CC1101

Neste apêndice são apresentados algumas características do *transceiver* RF CC1101 que permitem compreender melhor a implementação descrita no capítulo 5.

<table>
<tr><th>Bits</th><th>Name</th><th>Description</th></tr>
<tr><td>7</td><td>CHIP_RDYn</td><td>Stays high until power and crystal have stabilized. Should always be low when using the SPI interface.</td></tr>
<tr><td>6:4</td><td>STATE[2:0]</td><td>Indicates the current main state machine mode
<table>
<tr><th>Value</th><th>State</th><th>Description</th></tr>
<tr><td>000</td><td>IDLE</td><td>IDLE state
(Also reported for some transitional states instead of SETTLING or CALIBRATE)</td></tr>
<tr><td>001</td><td>RX</td><td>Receive mode</td></tr>
<tr><td>010</td><td>TX</td><td>Transmit mode</td></tr>
<tr><td>011</td><td>FSTXON</td><td>Fast TX ready</td></tr>
<tr><td>100</td><td>CALIBRATE</td><td>Frequency synthesizer calibration is running</td></tr>
<tr><td>101</td><td>SETTLING</td><td>PLL is settling</td></tr>
<tr><td>110</td><td>RXFIFO_OVERFLOW</td><td>RX FIFO has overflowed. Read out any useful data, then flush the FIFO with SFRX</td></tr>
<tr><td>111</td><td>TXFIFO_UNDERFLOW</td><td>TX FIFO has underflowed. Acknowledge with SFTX</td></tr>
</table></td></tr>
<tr><td>3:0</td><td>FIFO_BYTES_AVAILABLE[3:0]</td><td>The number of bytes available in the RX FIFO or free bytes in the TX FIFO</td></tr>
</table>

Figura A.1: Estrutura do *Status* byte[20].

Address	Strobe Name	Description
0x30	SRES	Reset chip.
0x31	SFSTXON	Enable and calibrate frequency synthesizer (if MCSM0.FS_AUTOCAL=1). If in RX (with CCA): Go to a wait state where only the synthesizer is running (for quick RX / TX turnaround).
0x32	SXOFF	Turn off crystal oscillator.
0x33	SCAL	Calibrate frequency synthesizer and turn it off. SCAL can be strobed from IDLE mode without setting manual calibration mode (MCSM0.FS_AUTOCAL=0)
0x34	SRX	Enable RX. Perform calibration first if coming from IDLE and MCSM0.FS_AUTOCAL=1.
0x35	STX	In IDLE state: Enable TX. Perform calibration first if MCSM0.FS_AUTOCAL=1. If in RX state and CCA is enabled: Only go to TX if channel is clear.
0x36	SIDLE	Exit RX / TX, turn off frequency synthesizer and exit Wake-On-Radio mode if applicable.
0x38	SWOR	Start automatic RX polling sequence (Wake-on-Radio) as described in Section 19.5 if WORCTRL.RC_PD=0.
0x39	SPWD	Enter power down mode when CSn goes high.
0x3A	SFRX	Flush the RX FIFO buffer. Only issue SFRX in IDLE or RXFIFO_OVERFLOW states.
0x3B	SFTX	Flush the TX FIFO buffer. Only issue SFTX in IDLE or TXFIFO_UNDERFLOW states.
0x3C	SWORRST	Reset real time clock to Event1 value.
0x3D	SNOP	No operation. May be used to get access to the chip status byte.

Figura A.2: Comandos *Strobe*[20].

Address	Register	Description
0x30 (0xF0)	PARTNUM	Part number for CC1101
0x31 (0xF1)	VERSION	Current version number
0x32 (0xF2)	FREQEST	Frequency Offset Estimate
0x33 (0xF3)	LQI	Demodulator estimate for Link Quality
0x34 (0xF4)	RSSI	Received signal strength indication
0x35 (0xF5)	MARCSTATE	Control state machine state
0x36 (0xF6)	WORTIME1	High byte of WOR timer
0x37 (0xF7)	WORTIME0	Low byte of WOR timer
0x38 (0xF8)	PKTSTATUS	Current GDOx status and packet status
0x39 (0xF9)	VCO_VC_DAC	Current setting from PLL calibration module
0x3A (0xFA)	TXBYTES	Underflow and number of bytes in the TX FIFO
0x3B (0xFB)	RXBYTES	Overflow and number of bytes in the RX FIFO
0x3C (0xFC)	RCCTRL1_STATUS	Last RC oscillator calibration result
0x3D (0xFD)	RCCTRL0_STATUS	Last RC oscillator calibration result

Figura A.3: Registos de estado[20].

	Write		Read		
	Single Byte	Burst	Single Byte	Burst	
	+0x00	+0x40	+0x80	+0xC0	
0x00	IOCFG2				R/W configuration registers, burst access possible
0x01	IOCFG1				
0x02	IOCFG0				
0x03	FIFOTHR				
0x04	SYNC1				
0x05	SYNC0				
0x06	PKTLEN				
0x07	PKTCTRL1				
0x08	PKTCTRL0				
0x09	ADDR				
0x0A	CHANNR				
0x0B	FSCTRL1				
0x0C	FSCTRL0				
0x0D	FREQ2				
0x0E	FREQ1				
0x0F	FREQ0				
0x10	MDMCFG4				
0x11	MDMCFG3				
0x12	MDMCFG2				
0x13	MDMCFG1				
0x14	MDMCFG0				
0x15	DEVIATN				
0x16	MCSM2				
0x17	MCSM1				
0x18	MCSM0				
0x19	FOCCFG				
0x1A	BSCFG				
0x1B	AGCCTRL2				
0x1C	AGCCTRL1				
0x1D	AGCCTRL0				
0x1E	WOREVT1				
0x1F	WOREVT0				
0x20	WORCTRL				
0x21	FREND1				
0x22	FREND0				
0x23	FSCAL3				
0x24	FSCAL2				
0x25	FSCAL1				
0x26	FSCAL0				
0x27	RCCTRL1				
0x28	RCCTRL0				
0x29	FSTEST				
0x2A	PTEST				
0x2B	AGCTEST				
0x2C	TEST2				
0x2D	TEST1				
0x2E	TEST0				
0x2F					
0x30	SRES		SRES	PARTNUM	Command Strobes, Status registers (read only) and multi byte registers
0x31	SFSTXON		SFSTXON	VERSION	
0x32	SXOFF		SXOFF	FREQEST	
0x33	SCAL		SCAL	LQI	
0x34	SRX		SRX	RSSI	
0x35	STX		STX	MARCSTATE	
0x36	SIDLE		SIDLE	WORTIME1	
0x37				WORTIME0	
0x38	SWOR		SWOR	PKTSTATUS	
0x39	SPWD		SPWD	VCO_VC_DAC	
0x3A	SFRX		SFRX	TXBYTES	
0x3B	SFTX		SFTX	RXBYTES	
0x3C	SWORRST		SWORRST	RCCTRL1_STATUS	
0x3D	SNOP		SNOP	RCCTRL0_STATUS	
0x3E	PATABLE	PATABLE	PATABLE	PATABLE	
0x3F	TX FIFO	TX FIFO	RX FIFO	RX FIFO	

Figura A.4: Espaço de endereçamento SPI do *transceiver* CC1101[20].

Apêndice B

Testes realizados

Aqui são apresentadas algumas imagens dos testes realizados no âmbito do capítulo 5.

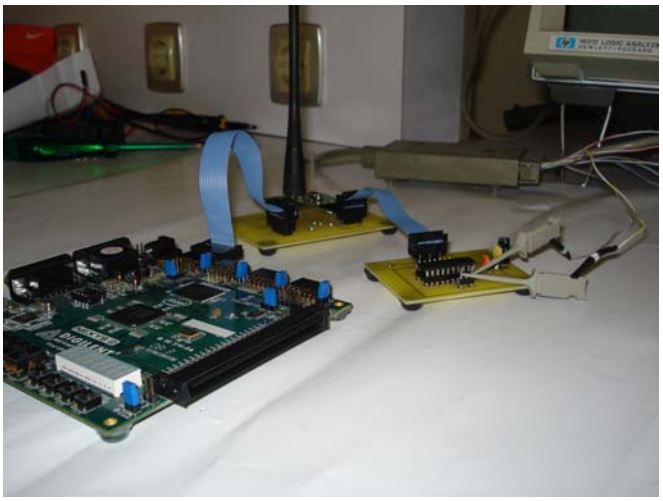


Figura B.1: Medição dos sinais SPI através de um analisador de lógica.

Figura B.2: Distância da ligação sem fios no teste 2 (240 metros).

Figura B.3: Distância máxima da ligação sem fios (270 metros).

Apêndice C

Lista de Acrónimos

ACC Adaptative Cruise Control
ADC Analog-to-Digital Converter
ASIC Application Specific Integrated Circuits
ASSP Application Specific Standard Product
ASK Amplitude-Shift Keying
CCA Clear Channel Assessment
CPLD Complex Programmable Logic Device
CRC Cyclic Redundancy Check
CS Chip Select
DAC Digital-to-Analog Converter
EEPROM Electrically Erasable Programmable Read Only
EPROM Erasable Programmable Read Only Memory
FIFO First In First Out
FPGA Field Programmable Gate Array
FSK Frequency-Shift Keying
FSM Finite State Machine
FEC Forward Error Correction
GFSK Gaussian Frequency-Shift Keying
HDL Hardware Description Language
HFSM Hierarchical Finite **S**tate Machine
IP Intellectual Property
ISE Integrated Software Environment
ISR Interrupt Service Routine
LFSR Linear Feedback Shift Register
LNA Low-Noise Amplifier

LQI Link Quality Indication
LUT Lookup Table
MISO Master Input Slave Output
MOSI Master Output Slave Input
MSK Minimum-Shift Keying
OOK On-Off Keying
PA Power Amplifier
PAL Programmable Array Logic
PC Personal Computer
PHFSM Parallel Hierarchical Finite State Machine
PLA Programmable Logic Array
PLD Programmable Logic Device
PROM Programmable Read Only Memory
RAM Random Access Memory
RISC Reduced Instruction Set Computer
ROM Read Only Memory
RSSI Received Signal Strength Indication
RTR Run-Time Reconfiguration
SCLK Serial Clock
SDI Slave Data In
SDO Slave Data Out
SoC System-on-Chip
SPI Serial Peripheral Interface
SPLD Simple Programmable Logic Device
SS Slave Select
USB Universal Serial Bus
VHDL VHSIC Hardware Description Language
WOR Wake-On-Radio

Bibliografia

[1] Valery Sklyarov; Iouliia Skliarova; Abílio Neves. Modeling and implementation of automatic system for garage control. *Proc. ICCAS-SICE, Fukuoka, Japão*, Agosto de 2009.

[2] Jari Nurmi. *Processor Design, System-on-Chip Computing for ASICs and FPGAs.* Springer, 2007.

[3] http://www.intel.com/pressroom/kits/events/moores_law_40th.

[4] Rahul Dubey. *Introduction to Embedded System Design Using Field Programmable Gate Arrays.* Springer, 2009.

[5] http://www.xilinx.com/support/documentation/application_notes/xapp213.pdf.

[6] http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf.

[7] Vikram K.N and V. Vasudevan. Hardware-software co-simulation of bus-based reconfigurable systems.

[8] Taeweon Suh; HsienHsin S. Lee; ShihLien Lu and Johh Shen. Initial observations of hardware/software cosimulation using fpga in architecture research.

[9] Derek Chiou; Dam Sunwoo; Joonsoo Kim; Nikhil Patil; William H. Reinhart; D. Eric Johnson and Zheng Xu. The fast methodology for high-speed soc/computer simulation.

[10] Eric S. Chung; Eriko Nurvitadhi; James C. Hoe; Babak Falsafi; Ken Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using fpgas.

[11] http://www.ieeta.pt/ skl/LecturesInTallinnSept2008.html.

[12] http://www.ami.ac.uk/courses/ami4233_mtecap/u03/index.asp.

[13] http://www.xilinx.com/publications/products/cpld/logic_handbook.pdf.

[14] http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm.

[15] http://www.actel.com/images/products/solutions/auto/car_cutaway.jpg.

[16] http://www.xilinx.com/itp/xilinx10/books/manuals.pdf.

[17] http://www.model.com/products/default.asp.

[18] http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=451&Prod=NEXYS2.

[19] http://ece.gmu.edu/coursewebpages/ECE/ECE448/S09/documentation/RC10df.

[20] http://focus.ti.com/lit/ds/symlink/cc1101.pdf.

[21] http://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen_ds512.pdf.

[22] http://focus.ti.com/lit/an/swra112b/swra112b.pdf.

[23] http://www.xilinx.com/products/devices.htm.

[24] http://download.intel.com/museum/Moores_Law/Articles Press_Releases /Gordon_Moore_1965_Article.pdf.

[25] http://www.xilinx.com/tools/microblaze.htm.

[26] Iouliia Skliarova. *Arquitecturas reconfiguráveis para problemas de optimização combinatória*. PhD thesis, Universidade de Aveiro, 2004.

[27] Pong P. Chu. *FPGA PROTOTYPING BY VHDL EXAMPLES - Xilinx Spartan 3 Version*. John Wiley & Sons, Inc., 2008.

[28] M. Koster and J. Teich. (self-)reconfigurable finite state machines: theory and implementation. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pages 559–566, 4–8 March 2002.

[29] V. Sklyarov. Hierarchical finite-state machines and their use for digital control. 7(2):222–228, June 1999.

[30] Tsutomu Maruyama; Masaaki Takagi and Tsutomu Hoshino. Hardware implementation techniques for recursive calls and loops. *P. Lysaght, J. Irvine, and R. Hartenstein (Eds.): FPL 99, LNCS 1673*, pages 450–455, 1999.

[31] G. Ferizis and H. E. Gindy. Mapping recursive functions to reconfigurable hardware. In *Proc. International Conference on Field Programmable Logic and Applications FPL '06*, pages 1–6, Aug. 2006.

[32] S. Ninos and A. Dollas. Modeling recursion data structures for fpga-based implementation. In *Proc. International Conference on Field Programmable Logic and Applications FPL 2008*, pages 11–16, 8–10 Sept. 2008.

[33] Valery Sklyarov. Fpga-based implementation of recursive algorithms. *Microprocessors and Microsystems 28*, pages 197–211, 2004.

[34] V. Skylarov, I. Skilarova, and B. Pimentel. Fpga-based implementation and comparison of recursive and iterative algorithms. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 235–240, 24–26 Aug. 2005.

[35] V. Sklyarov and I. Skliarova. Design and implementation of parallel hierarchical finite state machines. In *Proc. Second International Conference on Communications and Electronics ICCE 2008*, pages 33–38, 4–6 June 2008.

[36] Synthesizable VHDL code for PHFSM Online: http://www.ieeta.pt/skl/Research/Projects/ISE_Projects/ParallelHFSM.rar.

[37] www.sapteka.net/IntroductionToPLD.htm.

[38] http://labspace.open.ac.uk/mod/resource/view.php?id=360465.

[39] http://www.xilinx.com/company/gettingstarted/index.htm.

[40] http://www.achronix.com/.

[41] http://www.xilinx.com/products/v6s6.htm.

[42] http://www.eda experten.de/pdf/ModelSimDesignerDS60.pdf.

[43] http://focus.ti.com/docs/prod/folders/print/cc1101.html.

[44] http://www.maxim ic.com/quick_view2.cfm/qv_pk/4755.

[45] http://www.analog.com/en/rfif-components/short-range transceivers/products/index.html.

[46] http://focus.ti.com/docs/toolsw/folders/print/smartrftm studio.html.

[47] Valery Sklyarov; Iouliia Skliarova. Design and implementation of priority buffer for embedded systems.

[48] http://infinibandfpga.googlecode.com/svn/trunk/Physical/Idle_LFSR.vhd.

[49] http://sweet.ua.pt/ a16360/index.html.

[50] http://www.toyota.eu/06_Safety/03_understanding_active_safety/03_cruise_control.aspx.

[51] http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator_ds317.pdf.

[52] http://www.csgnetwork.com/gpsdistcalc.html.

[53] http://www.deetc.isel.ipl.pt/sistemastele/Pr2/arquivo/folhas%20de%20apoio/elips%C3%B3ides%20de%20Fresnel.pdf.